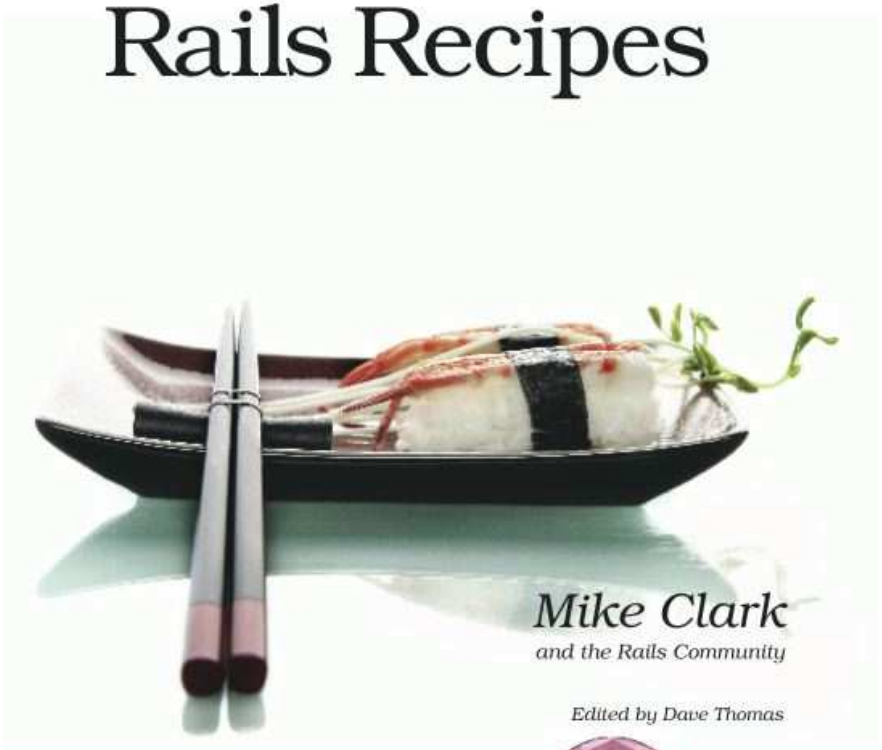
The Pragmatic Programmers

Advanced Rails Recipes



Mike Clark
and the Rails Community

Edited by Dave Thomas

The Facets of Ruby Series

The book you're reading is still under development. As an experiment, we're releasing this copy well before we normally would. That way you'll be able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned. The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long lines, incorrect hyphenations, and all the other ugly things that you wouldn't expect to see in a finished book. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Throughout this process you'll be able to download updated PDFs from your personal home page at http://www.pragprog.com/my_account (you'll need to create an account if you don't already have one). When the book is complete, you'll get the final version (and subsequent updates) from the same address. In the meantime, we'd appreciate you sending us your feedback at http://pragprog.com/titles/fr_arr/errata.

Thank you for taking part in this experiment.

► **Dave and Andy**

Advanced Rails Recipes

Mike Clark and the Rails Community

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 0-9787392-2-1

ISBN-13: 978-0-9787392-2-5

Printed on acid-free paper with 50% recycled, 15% post-consumer content.

B1.02 printing, January 2, 2008

Version: 2008-1-2

Contents

Chapter 1

Introduction

1.1 What Makes a Good Recipe Book?

If I were to buy a *real* recipe book—you know, a book about cooking food—I wouldn't be looking for a book that tells me how to dice vegetables or how to use a skillet. I can find that kind of information in an overview about cooking.

A recipe book is about how to *make* food you might not be able to easily figure out how to make on your own. It's about skipping the trial and error and jumping straight to a solution that works. Sometimes it's even about making food you never imagined you *could* make.

If you want to learn how to make great Indian food, you buy a recipe book by a great Indian chef and follow his or her directions. You're not just buying any old solution. You're buying a solution you can *trust* to be good. That's why famous chefs sell lots and lots of books. People want to make food that tastes good, and these chefs know how to make (and teach *you* how to make) food that tastes good.

Good recipe books *do* teach you techniques. Sometimes they even teach you about new tools. But they teach these skills within the context and with the end goal of *making something*—not just to teach them.

My goal for *Advanced Rails Recipes* is to teach you how to make great stuff with Rails and to do it right on your first try. These recipes (and the techniques they contain) are extracted from my own work and from the work of other "great chefs" of Rails around the community.

I also hope to show you not only *how* to do things but to explain *why* they work the way they do. After reading through the recipes, you

should walk away with a new level of Rails understanding to go with a huge list of successfully implemented hot new application features.

Not all of these recipes are full meals. To spice things up, I've included a number of smaller side dishes, which I've called *snacks*. Typically one or two pages long, these snacks will help satisfy those cravings we all get between meals.

1.2 What Makes This an Advanced Recipe Book?

Sushi is a treat for the eyes, as much as it's a treat for my taste buds. The sushi chefs behind the bar of my favorite local spot take great pride in making not only delicious dishes, but exquisite looking ones as well. And they seem to derive a great deal of satisfaction when their creations are enjoyed by me—a hungry programmer. Their goal isn't to have me stumble away from the table stuffed to the gills, but instead for me to leave pleasantly satisfied by the entire experience.

My goal for this book is similar. I don't want to load you up with heaps of cryptic, overly-clever code that sits heavy in your application. In my opinion, that's not what being advanced is about. It's actually the other way around: the programmers I admire strive to find elegant, practical solutions to complex problems. Indeed, making the code work is the easy part. Like the sushi chef, it's the presentation that takes a lifetime to master.

When the first *Rails Recipes* [Fow06] book was written, most of the Rails knowledge was concentrated in a small group of experts. These days, with new Rails applications being launched almost weekly and so many different problems being solved, the state of the art is spread across the community.

To accurately capture what specific problems advanced Rails developers are tackling, and how, we asked the Rails community to contribute their own special recipes. This book is an informal survey of what the best developers in the Rails community think is *advanced* and *important*.

1.3 Who's It For?

Advanced Rails Recipes is for people who understand Rails and now want to see how an experienced Rails developer would attack specific problems. Like with a real recipe book, you should be able to flip

through the table of contents, find something you need to *get done*, and get from start to finish in a matter of minutes.

When you're busy trying to *make* something you don't have spare time to read through introductory material. I'm going to assume you know the basics and that you can look up API details in a tutorial or an online reference. So if you're still in the beginning stages of learning Rails, be sure to have a copy of *Agile Web Development with Rails* [TH05] and a bookmark to the Rails API documentation handy.[1]

1.4 Who's Talking?

As this book is a compendium of tasty and unique recipes from the community of Rails developers, I've adopted a few conventions to keep the voice of the book consistent.

When a problem is being solved, *we're* doing it together—you, the reader, and the contributor of the recipe ("Let's build an ark, shall we?"). Then, when the contributor of the recipe needs to relay something about their experience, look for *I* and *my* ("I have a yellow rubber ducky on top of my monitor."). Lastly, if I want to comment on recipes that aren't my own, I'll include a "Mike says..." note.

1.5 Rails Version

The examples in this book, except where noted, require Rails 2.0 or higher. The Capistrano examples are based on Capistrano 2.0. I made no attempt to try them with older versions.

1.6 Resources

The Pragmatic Programmers have set up a forum for *Advanced Rails Recipes* readers to discuss the recipes, help each other with problems, expand on the solutions, and even write new recipes. You can find the forum by following the *Discussions* link from the book's home page at http://pragprog.com/titles/fr_arr.

The book's errata list is at http://pragprog.com/titles/fr_arr/errata. If you submit any problems you find, we'll list them there.

1. http://api.rubyonrails.org

You'll find links to the source code for almost all the book's examples at http://pragprog.com/titles/fr_arr/code.

If you're reading the PDF version of this book, you can report an error on a page by clicking the "erratum" link at the bottom of the page, and you can get to the source code of an example by clicking the gray lozenge containing the code's file name that appears before the listing.

1.7 Acknowledgments

Recipe contributions include the contributor's name and bio. Any errors or omissions are my own as the editor.

Thank you all for taking the time to share your tips and and tricks! It's been a lot of fun working with you. If your recipe didn't make it into this version of the beta book, don't despair. I'll be rolling more out incrementally as the book progresses.

Chad Fowler was the real inspiration behind this book. His original *Rails Recipes* [Fow06] book continues to be a source of practical *Get Er' Done!* techniques while I'm developing Rails apps. This book tries to follow the same style and format as the original. Thanks, Chad, for being an invaluable mentor throughout this process!

The Rails core team were very helpful in reviewing recipes. Thanks guys, for being a sounding board and patiently working through problems with me. And double-thanks to Jamis Buck for also reviewing Capistrano recipes.

Nicole makes everything possible. She encouraged me to compile and edit this book, knowing full well what that actually meant.

Most important, thank *you*, dear reader, for reading this book. It means a great deal to me that you would take the time. I hope we get to meet someday soon.

Mike Clark
December 2007
`mike@clarkware.com`

1.8 Tags and Thumb tabs

I've tried to assign tags to each recipe. If you want to find recipes that have something to do with Mail, for example, find the Mail tab at the edge of this page. Then look down the side of the book: you'll find a thumb tab that lines up with the tab on this page for each appropriate recipe.

Part I

REST and Routes Recipes

Recipe 1

Putting A Resource On The Web

Problem

You've heard all the buzz about creating RESTful this and that. There's little doubt that Rails 2.0 heavily favors the REST architectural style,[2] and will continue to.

You're feeling a little out of the loop, and what you've heard so far is a tad too academic. As a practical matter, you'd like create a web-accessible API for one of your models and, as a bonus, learn some conventions to help keep your controllers skinny. What can resources do for *you*?

Solution

Let's forget about REST for a moment and try to solve a problem. Let's say we organize events and we want a web-based interface for creating, reading, updating, and deleting events (folks in the know call that CRUD). Scaffolding is the quickest way to get from a database schema to the user. And in Rails 2.0, we can put up scaffolding in one fell swoop:

```
$ script/generate scaffold event name:string description:text ↩
                                price:decimal starts_at:datetime
```

That command generates a bunch of code: a controller with no less than seven actions, template files for actions that need them, and even a migration file with the database columns we asked for. The only thing left for us to do is create the database and apply the migration:

```
$ rake db:create
$ rake db:migrate
```

Now we fire up the application, and we have a full HTML interface to CRUD (the verb form) events. It sounds like the old Rails scaffolding, but there's a twist. You may have noticed that the following line was added to our config/routes.rb file:

2. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

```
map.resources :events
```

What that does is subtle, but significant: It dynamically adds a bunch of routes for accessing our events (called *resources*) via URLs. The routes map to the seven actions in our controller: index, show, new, create, edit, update, and destroy. We can see all the routes by typing:

```
$ rake routes
```

Let's look at a few just to get a taste of what's going on behind the scenes. First, we have routes for dealing with the events *collection*:

```
events GET  /events      {:action=>"index",  :controller=>"events"}
       POST /events      {:action=>"create", :controller=>"events"}
```

So, to list all our events—the index action—we'd send in the URL /events. And inside of our application we can use the events_url helper to generate a URL for listing the events. The exact same incoming URL is mapped to our create action. The only difference is the HTTP verb that's used: GET is a read-only operation that lists the events and POST is a write operation that creates a new event in our collection.

We also have routes for dealing with a specific *member* of the events collection:

```
event  GET  /events/:id  {:action=>"show", :controller=>"events"}
       PUT  /events/:id  {:action=>"update", :controller=>"events"}
```

Again, the URL is the same for both actions. The HTTP verb is used to disambiguate whether we want to read or update a single event by its primary key. Inside our application, we can use event_url(@event), for example, to generate a URL to show our event.

So, in summary, the map.resources line generates routes into our application using both the incoming URL *and* an HTTP verb. The immediate upshot is we can make any of our models into resources (events, registrations, users, etc.) and then manipulate them through a uniform URL scheme. It's just one more example of Rails conventions removing guesswork.

OK, that's all well and good. But browsers generally issue GET and POST requests, leaving the other HTTP verbs to the academics. So how do we tell our application that we want to update an existing event (a PUT) or delete an event (a DELETE)? Well, that involves a few more new conventions.

If we have an @event model object (and it's declared to be a resource), then in our new.html.erb and edit.html.erb forms, we can simply use:

```
<% form_for(@event) do |f| %>
```

The form_for will generate the appropriate form tag depending on the state of the @event object. If it's a new record, form_for generates this:

```
<form action="/events" method="post">
```

This will post to our create action because the HTTP verb and URL map to that action. However, if the event is an existing record, the form needs to post to the update action. To do that, it slaps in a hidden field to simulate a PUT operation which in turn triggers the proper route when Rails intercepts the request:

```
<form action="/events/1" method="post">
  <input name="_method" type="hidden" value="put" />
```

OK, so at the end of all this we're still managing events in the browser, just with special URLs. This gives our resources uniformity on the web and helps clean up a bunch of code inside our application.

Now let's say we'd like to introduce this application to another, and have them speak XML to each other. For that we turn our attention to the scaffold-generated index action. It has the familiar stuff, but it also sports a respond_to block:

Download Rest/app/controllers/events_controller.rb

```
def index
  @events = Event.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @events }
  end
end
```

Here's what's going on. The action that sets up a collection of events, and how they're rendered (the format) depends on what the client asks for. By default, browsers prefer HTML. In that case the action falls through the bottom and renders the index.html.erb template.

Here's where things get interesting. The URL conventions for CRUDing our event resources gives us a common lingo and respond_to gives us a way to vary how the resources are represented. So if our other application (the client program) wants to fetch a resource collection or member as XML, it simply tacks .xml to the end of the URL:

http://localhost:3000/events.xml

http://localhost:3000/events/1.xml

That gives us some XML back. Our client program might also want to create, update, or delete events. To do that, it needs to send a hunk of XML to the appropriate resource URL using the corresponding HTTP verb. But what we really want is a client-side program that does all that for us. After all, given the routing conventions used by map.resources it should be fairly easy to generalize remote access.

Enter Active Resource. It's a library that knows how to build URLs for accessing Rails resources, and it'll push XML over HTTP until the cows come home. Indeed, here's where having an application that responds to XML really shines.

First, we write a standalone Ruby program: the Active Resource client. It doesn't need to load Rails per se, but we do need to tell it where Active Resource lives on our file system:

Download Rest/services/event.rb

```
require File.join(File.dirname(__FILE__),
                  "../vendor/rails/activeresource/lib/active_resource")
```

We have a collection of event resources living in our server, so next we create a proxy class pointing to where its resource lives:

Download Rest/services/event.rb

```
class Event < ActiveResource::Base
  self.site = "http://localhost:3000"
end
```

Now the fun begins. All the standard CRUD-level operations are available, as if our proxy class was a real Active Record model. Let's find all the events and print their names:

Download Rest/services/event.rb

```
events = Event.find(:all)
puts events.map(&:name)
```

Perhaps we need to find a specific event and update its attributes:

Download Rest/services/event.rb

```
e = Event.find(1)
e.price = 20.00
e.save
```

And to round out the tour, we can also create and delete events:

Download Rest/services/event.rb

```
e = Event.create(:name => "Shortest event ever",
                 :starts_at => 1.second.ago)
```

```
e.destroy
```

A lot is happening behind the scenes. To see which URLs it's using, let's configure a logger:

Download Rest/services/event.rb

```
ActiveResource::Base.logger = Logger.new("#{File.dirname(__FILE__)}/ares.log")
```

Here's the output, which can be quite handy for debugging an onerous client:

```
GET http://localhost:3000/events.xml
--> 200  (3425b 0.04s)
GET http://localhost:3000/events/1.xml
--> 200  (378b 0.03s)
PUT http://localhost:3000/events/1.xml
--> 200  (1b 0.08s)
POST http://localhost:3000/events.xml
--> 201  (405b 0.03s)
DELETE http://localhost:3000/events/12.xml
--> 200  (1b 0.02s)
```

REST makes for good gravy, but it's not the main course. We have a full API for creating, reading, updating, and deleting events via the browser or a remote client program. And we have conventions—a consistent set of URLs that map to a consistent set of actions. In general, this makes deciding where things go a lot easier. In particular, we no longer have controllers that are dumping grounds for spurious actions.

Discussion

It's important to note that REST (map.resources in particular) and respond_to blocks are orthogonal. You can use respond_to without resources, and vice versa.

While Active Resource ships with Rails and uses some of its support classes, it's not necessarily Rails specific. You can use it to reach out to any server supporting the URL, HTTP verb, and XML format conventions of Rails.

Also See

The map.resources method only generates routes for the seven CRUD actions. This begs the question: How do I call actions outside of this set? Recipe 2, *Adding Your Own REST Actions (Or Not)*, on the following page shows how to customize the routes for application-specific concerns.

Recipe 2

Adding Your Own REST Actions (Or Not)

Problem

The RESTful style baked into Rails is starting to make some sense, and the conventions are working well for some things, but you just can't get your head wrapped around special cases. The devil's in the details, as they say. So when the seven CRUD actions of a resource seem to fall short, how (and more important *where*) do you deal with the edge cases?

Solution

Before we get started, I'll give you the bad news: There are no hard and fast rules we can apply in this recipe. This is largely a matter of software design, and design is all about trade-offs. That is to say, the real world is a wonderfully messy place and modeling it with straight-line boxes and arrows is anything but exact. REST, on the other hand, is an idyllic world. Our job is to find the common ground for the good of our application and its users.

Let's walk through three examples where REST commonly trips us up: sorting, searching, and nested resources. Then we'll step back and see if we can tease out some guidelines.

First, we have a collection of event resources that we'd like to sort by name or start time. To do that we *could* define a new sort action, for example, and punch a hole in the routes for it. But presenting a sorted list of events is no different than what an index action does. So all we need to do here is let our existing index action handle sorting:

Download Rest/app/controllers/events_controller.rb

```
def index
  sort_by = (params[:order] == 'starts_at' ? 'starts_at desc' : 'name')
  @events = Event.find(:all, :order => sort_by)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @events }
```

```
  end
end
```

Then to sort events, we send in this URL, for example:

http://localhost:3000/events?order=starts_at

Indeed, that URL is a resource in its own right: it uniquely identifies a collection of events.

Next we'd like to search for events given a search term, such as the event name. Similar to sorting, we could send a term parameter in to the index method. However, our search implementation has slightly different concerns. For example, we might want to render search results with rankings, a reminder of the term we used, and a "Did you really mean…" tip. Hrm, search doesn't fit as neatly into the index box.

In this particular case, let's create a new search action in our EventsController:

Download Rest/app/controllers/events_controller.rb

```
def search
  @term = params[:term]
  @events = Event.search(@term)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @events }
  end
end
```

The routes generated by map.resources :events don't know about that action. So we need to add an extension to the map.resources call in config/routes.rb:

```
map.resources :events, :collection => { :search => :get }
```

Now we have a route to the search action, accessible via a GET request. It applies all the events—the collection of resources. To get there we navigate to:

http://localhost:3000/events/search?term=rubyconf

Again, this is a totally RESTful URL. You can think of search as a subresource of the events resource.

Finally, we'd like people to be able to register for events. At first blush, it appears we need a register action on the EventsController. To do that

though means we miss a design opportunity. We're really missing a key resource here: registrations.

So rather than polluting the EventsController, let's just make registrations a nested-resource of events by revising our routes.rb file:

```
map.resources :events,
  :collection => { :search => :get },
  :has_many => :registrations
```

Then we need some way to register for an event, so we'll add this to an event page that has a event variable already set up:

```
<%= button_to "Register", event_registrations_path(:event_id => event) %>
```

This will post to the create action in our RegistrationsController, which might then look something like this, for example:

Download Rest/app/controllers/registrations_controller.rb

```
class RegistrationsController < ApplicationController

  def create
    @event = Event.find(params[:event_id])
    if @current_user.register_for(@event)
      flash[:notice] = "Thanks for registering!"
    else
      flash[:notice] = "You're already registered for that event!"
    end
    @events = @current_user.events
  end

end
```

OK, let's step back. In the first example we extended an existing action and in the second example we extended the routes. So, should searching always be a new action? Well, it depends. If the search templates and everything else are similar enough to index, then it's perfectly acceptable (in the eyes of REST) to implement searching as a variant of the index action. If searching is fundamentally different, then it may well deserve its own action. Finally, in the third example, looking at our application through the REST lens unveiled a missing resource. So if there's a lesson here it's that resources serve as expert guides, if only we listen to them.

Recipe 3

Authenticating REST Clients

Problem

You're developing a RESTful application. Perhaps it's an event management system with user accounts. Naturally, you need to protect access to resources of the site with a login and password. You've done this before, you know, back in the old days. But this is a new day, so how do you authenticate users in a RESTful way?

Solution

Let's assume we already have a User model. To check if a user exists for a given login and password, we can call:

```
User.authenticate(login, password)
```

Given that, we need a form to accept the login and password. And of course we need a couple controller actions to pop the form and authenticate the user. Now, we could slap those actions in any old controller, but REST is always asking the question: What's the resource?

The thing we're really trying to manage when dealing with web-based authentication is an HTTP session. It's the resource that knows whether a user is currently logged in or not.

So let's start by adding the RESTful routes for a session to our config/routes.rb file:

```
map.resource :session
```

Note that we're using map.resource (singular) here, not plural as is usually the case. For a given user, we only need one session. The singular form generates routes and helpers using the singular name (session) as we'll see in minute.

Next we need to create a SessionsController for the session. (Resource controllers are always plural.) The new action pops the form, and the create action stashes the user's id in the session if the user logs in successfully:

Download Rest/app/controllers/sessions_controller.rb

```
def new
end
```

```
def create
  user = User.authenticate(params[:login], params[:password])
  if user
    session[:user_id] = user.id
    flash[:notice] = "Welcome back, #{user.login}!"
    redirect_to events_url
  else
    flash[:error] = "Invalid email/password combination!"
    render :action => 'new'
  end
end
```

Then when the user logs out, we delete their session in the destroy action:

Download Rest/app/controllers/sessions_controller.rb

```
def destroy
  reset_session
  flash[:notice] = "You've been logged out."
  redirect_to new_session_url
end
```

This is all pretty standard authentication stuff, with the exception of being able to call the new_session_url to generate the URL back to the new action. To invoke these actions, we'll of course need some links:

```
<%= link_to 'Login', new_session_path  %>
<%= link_to 'Logout', session_path, :method => :delete %>
```

Next we need a login form for the new action:

Download Rest/app/views/sessions/new.html.erb

```
<% form_tag session_path do -%>
<fieldset>
  <p>
    <label for="login" class="required">Login</label>
    <%= text_field_tag :login, params[:login] %>
  </p>
  <p>
    <label for="password" class="required">Password</label>
    <%= password_field_tag :password, params[:password] %>
  </p>
  <p>
    <%= submit_tag 'Log in' %>
  </p>
</fieldset>
<% end -%>
```

This is a standard form, but notice that it uses the session_path helper in the form_tag. In this case the form will issue a POST to /session. And

according to the routes that were added by map.resource, that URL and verb pair always maps to the create action.[3]

At this point we have the session resource all ready to go, and a way for users to log in and out. But we make the rules around here about when a login is required. If there's an id in session[:user_id], we know the user is logged in. If they aren't logged in and they try accessing our site, we need to issue a redirect to the login form. Easy enough. We just add a before_filter in the ApplicationController:

Download Rest/app/controllers/application.rb

```
before_filter :login_required
```

Except we don't want to force a login when indeed the user is trying to log in (makes for angry users), so we'll just skip the before filter in the SessionsController:

Download Rest/app/controllers/sessions_controller.rb

```
skip_before_filter :login_required
```

Then we need to write the login_required method called by the before_filter:

Download Rest/app/controllers/application.rb

```
def login_required
  unless session[:user_id]
    flash[:notice] = "Please log in"
    redirect_to new_session_url
  end
end
```

OK, that gives us a way to log in via a browser in a RESTful way, and have the app remember that we're logged in. So far so good. But that's just one half of the equation.

Remember that our resources can also represent themselves as XML, for example. (That's what the respond_to blocks in our controllers are all about.) And let's say we have an Active Resource client that uses XML to chat with the resources exposed by our application. In this case there is no browser. So how do we authenticate this client program?

The web already has the answer: HTTP basic authentication. An encoded login and password are slipped in the HTTP headers, and away we go. So in our Active Resource client, we include the login and password as part of the URL pointing to where the resource lives:

3. Run rake routes for a peek inside.

```
class Event < ActiveResource::Base
  self.site = "https://mike:secret@localhost:3000"
end
```

Note that if we were to use http://, the encoded login and password will travel across the 'net in plaintext which any hacker worth his salt can decode over lunch, so don't forget to use https://! And while we're talking about security, let's go ahead and use the HighLine[4] library to prompt for a login and password, rather than hard-coding them in our client:

Download Rest/services/event_with_auth.rb

```
require 'rubygems'
require 'highline/import'

def prompt(prompt, mask=true)
  ask(prompt) { |q| q.echo = mask}
end

def login
  prompt('Login: ')
end

def password
  prompt('Password: ', '*')
end

class Event < ActiveResource::Base
  self.site = "https://#{login}:#{password}@localhost:3000"
end
```

Neat and tidy. We almost have all the ingredients mixed, I promise. We just need to fix up our server to handle HTTP basic authentication. Remember, our login_required filter just checks for a user id in the session. That won't work for our Active Resource client because it's sending credentials in HTTP headers (and it doesn't have cookies to store session data in). So as the final step we need to spiff up the login_required filter method to handle both types of clients:

Download Rest/app/controllers/application.rb

```
def login_required
  respond_to do |format|
    format.html do
      if session[:user_id]
        @current_user = User.find(session[:user_id])
      else
        flash[:notice] = "Please log in"
```

4. http://rubyforge.org/projects/highline/

```
        redirect_to new_session_url
      end
    end
    format.xml do
      user = authenticate_with_http_basic do |login, password|
        User.authenticate(login, password)
      end
      if user
        @current_user = user
      else
        request_http_basic_authentication
      end
    end
  end
end
```

There's our old friend respond_to again. If the client wants HTML (it's the browser knocking), then we check the session. If the client wants XML (hello, Active Resource client), then we call the authenticate_with_http_basic method. It decodes the HTTP headers for us, and passes the login and password as block parameters. Then we just try to authenticate the user. If we find a matching user, we're good to go. Otherwise, we send a request back to the client to retry using the request_http_basic_authentication method.

Whew! That involved quite a few steps. Here's the good news: We now have all the authentication we need for all RESTful client types.

Discussion

Although several authentication libraries are available as plugins and generators, simple authentication is so easy to roll by hand that it's often not worth carrying around the extra baggage of a third-party plugin. Most importantly, by writing your own you will *understand* how it works. That way, when it comes time to debug what's going on, you'll be in good shape to get it done quickly.

Having said that, if you want an example of a slightly more complex authentication approach, check out Rick Olson's restful_authentication plugin.[5] In addition to creating a RESTful session environment, it can also generate everything you need to get started with users including an activation step.

5. http://svn.techno-weenie.net/projects/plugins/restful_authentication/

Also See

- See Recipe 1, *Putting A Resource On The Web*, on page 15 for an introduction to REST.

Custom Response Formats

By Patrick Reagan (http://www.viget.com)
Patrick is a recovering PHP user who finally realized the immense power that Rails brings to the web development space. As the Director of Application Development for Viget Labs, he's been helping lead the charge in adopting Rails as the framework of choice when building applications for their startup clients.

Problem

Rails knows about several pre-defined formats for responding to requests: HTML, JavaScript, XML, RSS, and so on. But how do you create your own formats?

Solution

Say we want to build an app to download or stream MP3 files we find online. Let's start by making it a RESTful application using scaffolding:

```
$ script/generate scaffold mp3 title:string url:string length:string
$ rake db:migrate
```

So now that we can manage MP3 resources, let's drop a few files in via the console:

```
$ ruby script/console
>> Mp3.create(:title => 'RoR Podcast: Chad Fowler',
   :url => 'http://paranode.com/~topfunky/audio/2005/Chad-Fowler.mp3',
   :length => "2747625"
=> #<Mp3 id: 1, ...>
>> Mp3.create(:title => 'RoR Podcast: Dave Thomas and Mike Clark',
  :url =>
'http://paranode.com/~topfunky/audio/2006/rails-032-thomas-and-clark.mp3',
  :length => "26664043")
=> #<Mp3 id: 2, ...>
```

Now let's turn our attention to the Mp3sController. Currently the show action only knows how to render our MP3 information as HTML or XML:

```
def show
  @mp3 = Mp3.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.xml  { render :xml => @mp3 }
  end
end
```

We also want to serve up playable audio when the show action is invoked. To do that, we need to register our MIME types in the config/initializers/mime_types.rb file that's included by default in all new applications:

Download RespondToFormats/config/initializers/mime_types.rb

```
Mime::Type.register 'audio/mpeg',    :mp3
Mime::Type.register 'audio/mpegurl', :m3u
```

So, when the browser requests the .mp3 or .m3u formats, our application needs to set the Content-Type header to the corresponding MIME type when it sends the response. We'll rely on the browser to handle the MIME type in the response properly.

Now, back in our controller we can add our formats to the show action's respond_to block:

Download RespondToFormats/app/controllers/mp3s_controller.rb

```
def show
  @mp3 = Mp3.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.xml { render :xml => @mp3 }
    format.mp3 { redirect_to @mp3.url }
    format.m3u { render :text => @mp3.url }
  end
end
```

Here's what happens: When a user requests the .mp3 format for download, we redirect to the MP3 file's URL. If the user requests the .m3u format to stream the MP3, we respond with a text file that includes a pointer to an actual MP3 resource. Most modern audio applications respect the M3U format and will "stream" the referenced resource by both downloading and playing the MP3 simultaneously.[6]

Let's give this a shot. Rails includes a default route to let us set the format in the URL. To download the first MP3, we use:

http://localhost:3000/mp3s/1.mp3

And to stream the first MP3 into our audio player, we use:

http://localhost:3000/mp3s/1.m3u

To add the appropriate links to our views, we can use named routes:

6. For me, this starts up iTunes and begins streaming the file. For Windows users, this action will typically open Windows Media Player, Winamp, or another configured application.

Download RespondToFormats/app/views/mp3s/show.html.erb

```
<p>
  <%= h @mp3.title %>
  (<%= link_to 'Download', formatted_mp3_url(@mp3, :mp3) %> |
   <%= link_to 'Stream',   formatted_mp3_url(@mp3, :m3u) %>)
</p>
```

This is a good start, but let's take it a step further. Right now we're streaming files one-by-one. That's not always convenient. So let's also allow a user to queue multiple MP3 streams using the playlist (PLS) file format. We'll go through the same steps to register the new MIME type:

Download RespondToFormats/config/initializers/mime_types.rb

```
Mime::Type.register 'audio/x-scpls', :pls
```

Because an audio playlist is essentially a listing of audio files, we serve it up through the index action:

Download RespondToFormats/app/controllers/mp3s_controller.rb

```
def index
  @mp3s = Mp3.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @mp3s }
    format.pls  { render :layout => false } # index.pls.erb
  end
end
```

All we need to do now is add the corresponding template to render the playlist in the proper format:

Download RespondToFormats/app/views/mp3s/index.pls.erb

```
[playlist]
NumberOfEntries=<%= @mp3s.length %>
<% @mp3s.each_with_index do |mp3, index| -%>
  File<%=   index + 1 %>=<%= h mp3.url %>
  Title<%=  index + 1 %>=<%= h mp3.title %>
  Length<%= index + 1 %>=<%= h mp3.length %>
<% end -%>

Version=2
```

The naming here is important. The format we're sending back is pls and we're using the ERB templating system to render the format. So the template file is called index.pls.erb, and it's only rendered if the pls format is requested.

With this plumbed in, a user can queue up both files we've added to our application using this URL:

http://localhost:3000/mp3s.pls

Discussion

These custom formats aren't limited to audio—you can add your own formats to serve up calendar files or anything that has a Content-Type recognized by a client application.

Catch All 404s

Problem

You want a record of all URLs that trigger 404s in your application, perhaps to plug holes in your routing scheme or identify legacy URLs you forgot to handle.

Solution

Let's follow a stray request through our application, writing code as we go. It starts with an incoming URL that doesn't map to any action in our application:

http://railsrecipes.com/please/catch/me

Ah, poor thing. Let's catch it by creating this route:

Download buffet/config/routes.rb

```
map.connect '*path', :controller => 'four_oh_fours'
```

Two things make this a catch-all route: it's the last route in the config/routes.rb file and it uses an asterisk to sponge up the incoming URL path parts. So for the example URL above, the path parameter would end up containing the array:

```
["please", "catch", "me"]
```

Now we need the FourOhFoursController to handle all the stray requests:

Download buffet/app/controllers/four_oh_fours_controller.rb

```
class FourOhFoursController < ApplicationController

  def index
    FourOhFour.add_request(request.host,
                           request.path,
                           request.env['HTTP_REFERER'] || '')

    respond_to do |format|
      format.html { render :file => "#{RAILS_ROOT}/public/404.html",
                           :status => "404 Not Found" }
      format.all  { render :nothing => true,
                           :status => "404 Not Found" }
    end

  end
```

```
end
```

There's not much to this controller. We just strip out a few interesting bits of the incoming request: the hostname (railsrecipes.com), the path (/please/catch/me), and the URL of the page that triggered this request if it exists. Before rending an appropriate 404 response back to the client, the FourOhFour model squirrels the request information away in the database so we have a permanent record:

Download buffet/app/models/four_oh_four.rb

```
class FourOhFour < ActiveRecord::Base

  def self.add_request(host, path, referer)
    request = find_or_create_by_host_and_path_and_referer(host, path, referer)
    request.count += 1
    request.save
  end

end
```

That just leaves us with creating the migration file, with some indices:

Download buffet/db/migrate/003_create_four_oh_fours.rb

```
class CreateFourOhFours < ActiveRecord::Migration

  def self.up
    create_table :four_oh_fours do |t|
      t.string  :host, :path, :referer
      t.integer :count, :default => 0

      t.timestamps
    end
    add_index :four_oh_fours, [:host, :path, :referer], :unique => true
    add_index :four_oh_fours, [:path]
  end

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

Then just create a listing of all 404s somewhere on the admin side of your app, and you've got yourself a convenient 404 report.

Discussion

But wait, there's more! You could also use the catch-all route to actually handle requests that don't map to a specific action. Say, for example, you have a database table that stores "pages": a URL path and the content to display when that URL is accessed. (You might even be calling this a *content management system*.) By modifying the controller just slightly, the catch-all route gives users access to all your content:

```
def index
  @page = CmsPage.find_by_path(params[:path])
  if @page
    render :inline => @page.body
  else
    # treat it as a 404
  end
end
```

Part II

Search Recipes

Improve SEO with Dynamic Meta Tags

Thanks to Dan Benjamin for the idea for this recipe.

Setting the HTML meta tags consistently across your app helps search engines guide people to your site. A good start is to put relevant titles, descriptions, and keywords on each page for the search engines to snarf up. It doesn't guarantee better rankings, but it sure can't hurt to use the HTML markup the way it was intended.

With a few instance variables and simple helpers, we can quickly add meta-tag consistency in the top of our application layout, like so:

Download ImproveSEOWithDynamicKeywords/app/views/layouts/application.html.erb

```
<%= page_title %>
<%= meta "description", meta_description %>
<%= meta "keywords", meta_keywords %>
```

Let's visit each helper method in turn.

Search engines love a good title tag, and it helps people using your site, too. The page_title helper creates an appropriate title for each page:

Download ImproveSEOWithDynamicKeywords/app/helpers/application_helper.rb

```
def page_title
  title = @page_title ? "| #{@page_title}" : ''
  %(<title>Bookstore #{title}</title>)
end
```

Now we just need to set the @page_title instance variable when we don't want the default page title. It turns out that a layout template has access to all the instance variables that are set in the templates and views that were used to render the page. So, if a page is showing a book, in the view we set @page_title to the book's title:

Download ImproveSEOWithDynamicKeywords/app/views/books/show.html.erb

```
<% @page_title = @book.title %>
```

Next come the meta tags. Let's start with a little helper that takes the meta-tag name and its content, then generate the actual HTML:

Download ImproveSEOWithDynamicKeywords/app/helpers/application_helper.rb

```
def meta(name, content)
```

```
  %(<meta name="#{name}" content="#{content} />")
end
```

That lets us write this in our layout:

```
<%= meta "description", "A Book" %>
```

Except if the current page is showing a book (we have an @book instance variable set), then the description should include the title and author. So next we write a helper just to generate the description meta-tag content:

Download ImproveSEOWithDynamicKeywords/app/helpers/application_helper.rb

```
def meta_description
  if @book and !@book.new_record?
    "Information about #{@book.title} by #{@book.author}."
  else
    "Books for programmers by programmers"
  end
end
```

Then we can mix the two together to get this:

```
<%= meta "description", meta_description %>
```

Last, but by no means least, let's go for broke by filling in the keywords meta tag with some of the book's attributes:

Download ImproveSEOWithDynamicKeywords/app/helpers/application_helper.rb

```
def meta_keywords
  if @book and !@book.new_record?
    [@book.title,
     @book.author,
     "#{@book.edition.ordinalize} Edition",
     @book.isbn,
     @book.pubdate.to_s(:month_year)].join(',')
  else
    %w(books programmers).join(',')
  end
end
```

Then back in the layout file it looks like this:

```
<%= meta "keywords", meta_keywords %>
```

Now we can rest easy knowing that all the book pages have meaningful meta tags. Here's an example:

```
<title>Bookstore | Agile Web Development with Rails</title>
<meta name="description" content="Information about Agile Web Development
                                  with Rails by Dave Thomas." />
<meta name="keywords" content="Agile Web Development with Rails,
```

```
Dave Thomas,2nd Edition,
978-0-9776166-3-3,Nov 2007" />
```

Recipe 7

Full-Text Search with Ferret

By Gregg Pollack (http://www.railsenvy.com)
This recipe is a blend of ideas, code, and text contributed independently by Mike Subelsky (http://www.subelsky.com/) and Gregg Pollack. I mixed them together to bring you a dish using the freshest ingredients.

Problem

At some point you're bound to need a search field. While single column searches over a few thousand records is easy, things slow down when you start searching millions of records across multiple database tables and columns. All the big boys use optimized search applications, so why not you?

Ingredients

- The ferret gem:

```
$ gem install ferret
```

- The acts_as_ferret plugin:

```
$ script/plugin install script/plugin install ↩
    svn://projects.jkraemer.net/acts_as_ferret/tags/stable
```

Solution

Admittedly there's more than one way to skin this cat, but Ferret[7] is a lightweight, and yet high performance, text search engine derived from the well-known Java Lucene project (which is what all the Java big boys use). And the acts_as_ferret plugin gives us a simple interface so we can start creating complex search indexes before the database melts.

Let's say we're running a job hunting site. People like to search through job postings, if only to remind themselves that they know more acronyms than the recruiters who post the jobs. Here's what our database schema looks like:

Download ActsAsFerret/db/migrate/001_create_job_postings.rb

```
create_table :job_postings do |t|
  t.string :name
```

7. http://ferret.davebalmain.com/trac

```
  t.text   :requirements, :description
end
```

The first thing we need to do is mix the acts_as_ferret goodies into our JobPosting model:

Download ActsAsFerret/app/models/job_posting.rb

```
class JobPosting < ActiveRecord::Base
  acts_as_ferret :fields => [ :name, :description, :requirements ],
                 :remote => true
end
```

Ferret can rapidly search through data because it builds optimized indexes on all search terms. If we don't specify which fields we'd like Ferret to index, it assumes we want to index all fields. This can be expensive, so instead we tell Ferret exactly which fields in our model we want to index (and do searches on).[8]

Take special note of the :remote => true option. If you leave that option out in production, you're playing with fire. Ferret stores its indices on the file system where the server is running, subdivided by environment. (By default this is RAILS_ROOT/index/development/job_posting, for example.) If you have multiple Rails processes running index operations on that directory, you'll quickly wind up with a corrupted index. Using :remote => true causes your model to connect to a remote Ferret server for index-friendly search operations. Let's set that up next.

The acts_as_ferret plugin automatically rolled out a configuration file for the remote Ferret server in config/ferret_server.yml:

```
production:
  host: ferret.yourdomain.com
  port: 9010
  pid_file: log/ferret.pid
development:
  host: localhost
  port: 9010
  pid_file: log/ferret.pid
test:
  host: localhost
  port: 9009
  pid_file: log/ferret.pid
```

8. You generally only want to index fields that have varying text content. It doesn't make sense to index fields that could be searched with less-expensive comparison searches that can already be done via Active Record's find method.

By default the development and test sections are commented out. If acts_as_ferret can't find a configuration for its current environment, then it just won't use a remote server. We don't necessary *need* a remote server unless we're in production, but it's good to test everything out locally anyway. So go ahead and uncomment all the environments. You may want to change the name of the pid_file for the three environments so that you can run servers for each environment simultaneously.

Then, to start the Ferret server, we just run:

```
$ script/ferret_server start
```

This will launch a distributed Ruby (dRuby) server that runs as a separate process on the configured port. And if all goes well, we see:

```
starting ferret server...
```

OK, let's run some searches in the console. First we'll create a couple of job postings:

```
$ ruby script/console
>> JobPosting.create(:name => "Rails Hacker",
              :description => "We have a foosball table",
              :requirements => "At least 10 years of Rails experience")
>> JobPosting.create(:name => "Ruby Hacker",
              :description => "We've seen the light",
              :requirements => "Can you explain what a symbol is?")
```

We have a couple search methods at our fingertips, the first of which is find_id_by_contents:

```
>> total, jobs = JobPosting.find_id_by_contents('hacker')
=> [2, [{:score=>0.029847851023078, :data=>{}, :model=>"JobPosting", :id=>"1"},
        {:score=>0.029847851023078, :data=>{}, :model=>"JobPosting", :id=>"2"}]]
>> total
=> 2
```

So we have two matches in the total variable, and in the jobs array we get the results with the ids and search scores for each of them. Using find_id_by_contents is good for times when we don't want to fetch all the matching objects from our database, but instead we just want to get their IDs and relevance scores. Then we could selectively pick the best results and how many we'll display. But let's say we want to go ahead and display the job posting information. This is where find_by_contents comes in:

```
>> results = JobPosting.find_by_contents('hacker')
=> #<ActsAsFerret::SearchResults:0x193ab24 @total_pages=1,
   @results=[
     #<JobPosting id: 1, name: "Rails Hacker",
```

```
      description: "We have a foosball table",
      requirements: "Must have at least 10 years of Rails experience">,
    #<JobPosting id: 2, name: "Ruby Hacker",
      description: "We've seen the light",
      requirements: "Can you explain what a symbol is?">],
  @per_page=2, @current_page=nil, @total_hits=2>
```

The neat thing about this is the way it slurps in our JobPosting objects. Calling find_by_contents turns around behind the scenes and calls find_id_by_contents to get all the ids from the search server. Then it issues one database query to select all job postings in that set of ids.

That gives us everything we need to run efficient searches. All that's left is putting it on the web. That's the easy part. Imagine we have a search action like this:

Download ActsAsFerret/app/controllers/job_postings_controller.rb

```
def search
  @results = JobPosting.find_by_contents(params[:term]).
               sort_by(&:ferret_rank)
end
```

Here we're sorting by ferret_rank, which refers to the sort order Ferret recommends for the hits, based on the relevance score. So then we could render our job postings in some meaningful order and show the score:

Download ActsAsFerret/app/views/job_postings/search.html.erb

```
<h1><%= pluralize(@results.size, 'result') %></h1>

<ul>
<% for job in @results %>
  <li><%= job.ferret_score %>: <%= link_to job.name, job %></li>
<% end %>
</ul>
```

Finally, to stop the remote Ferret server, we use:

```
$ script/ferret_server stop
```

Now, back to searching for those humorous job postings...

Discussion

The find_id_by_contents and find_by_contents methods take two optional parameters after the query string which can greatly extend the power of your queries: limit and offset. These should look familiar, and feel just like your normal ActiveRecord find searches.

Mike says...

I Once Had a Ferret

That's not entirely true, but I once needed a ferret. When we were building the Pragmatic Store we kept delaying the search implementation. It couldn't take long, right? Then a couple weeks before the launch, we decided we'd better spike it. So we opened up the contributions for this Ferret recipe and had it all working in a matter of minutes. We basically haven't touched it since. The critical ingredient to running Ferret in production is the use of the remote Ferret server.

Other acts_as_ferret options allow you to search multiple models at once and share indices among models.

Also See

- The acts_as_searchable plugin adds full-text searching capabilities based on Hyper Estraier.
- The acts_as_sphinx plugin adds full-text searching to Sphinx.
- The acts_as_tsearch plugin gives you access to Postgres' T-search native text indexing extension.
- MySQL has full-text indexing, as long as you use MyISAM tables.

Recipe 8

Active Record on Solr

By Erik Hatcher (http://code4lib.org/erikhatcher)
Erik, besides lamely teasing with rucene and ruby-lucene for years, co-authored *Lucene in Action* (**?**). He speaks passionately around the world on varying technical topics of interest, most recently on this very combination of Solr and Ruby at RailsConf and rubyconf 2007. Erik has worked at the University of Virginia's Applied Research in Patacriticism to shine Solr's light onto the 19th century world of art and poetry, and now works full-time on Lucene and Solr technologies for a startup company.

Problem

Your content needs to be full-text searchable (not in the puny SQL LIKE %whatever% way), and you want to allow what's called *faceted browsing* to help users narrow down results into meaningful categories. At the same time, you want Active Record creates, updates, and deletes to "just work" with the least amount of work.

Ingredients

- The Java Runtime Environment (JRE) 1.5
- The acts_as_solr plugin:

```
$ script/plugin install ↩
    svn://svn.railsfreaks.com/projects/acts_as_solr/trunk
```

Solution

We're going to pull out the big cooking utensils in this recipe, because we want a solution that's fast and scalable (and we set the bar unreasonably high by calling SQL "puny" in the problem statement). Solr[9] is an open source search server based on the tried and true Lucene[10] search library. Solr was created for CNET and open sourced to an active community—it's gained mindshare by the bulk of the expert information retrieval folks in the world. Solr runs in a Java servlet container, but don't worry about that, the acts_as_solr plugin comes with one to get us going in a jiffy.

9. http://lucene.apache.org/solr/
10. http://lucene.apache.org

Great, so let's search some books, shall we? We'll have books and categories in a has_and_belongs_to_many embrace. The only interesting migration file is for the books table. Here's what we can search on:

Download ActsAsSolr/db/migrate/001_create_books.rb

```
class CreateBooks < ActiveRecord::Migration

  def self.up
    create_table :books do |t|
      t.string :title, :asin, :author, :publisher
      t.date   :published_date
    end
  end

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

With the acts_as_solr plugin installed, we can go ahead and add Solr support to our Book model by (you guessed it) adding a acts_as_solr declaration:

Download ActsAsSolr/app/models/book.rb

```
class Book < ActiveRecord::Base
  acts_as_solr
  has_and_belongs_to_many :categories
end
```

Now it's time to fire up the Solr server. When we installed the acts_as_solr plugin, it added a handful of Rake tasks to manage the server. Here it goes:

```
$ rake solr:start
```

The plugin also dropped a config/solr.yml file that tells Solr which URL and port to run on. Similarly, it tells our Book model how to summon the mighty search server. The default development port is 8982; it's 8983 for production. So we can walk right up to Solr as a test:

http://localhost:8982/solr/

That wasn't so bad. Now we're ready to add a book and full-text search it. The easiest way to experiment with searching is using the console. We'll start by adding a book and its categories:

```
$ ruby script/console
>> book = Book.new(:title => "Solr Recipes",
                   :published_date => Date.today,
                   :publisher => "See The Light Publishing")
```

```
>> book.categories << Category.new(:name => "Yummy")
>> book.categories << Category.new(:name => "Information Retrieval")
>> book.save
```

Then, still in the console, we can use the Book.find_by_solr method to find all the books with "recipe" in any column:

```
>> results = Book.find_by_solr("recipe")
=> #<ActsAsSolr::SearchResults:0x2552ea8 @solr_data=
{:total=>1, :docs=>[#<Book:0x2552db8 @attributes={"title"=>"Solr
Recipes", "author"=>nil, "asin"=>nil, "id"=>"3", "publisher"=>nil,
"published_date"=>nil}>], :max_score=>0.73360884}>
>> results.docs
=> [#<Book id: 1, title: "Solr Recipes", asin: nil, author: nil,
publisher: "See The Light Publishing", published_date: "2007-12-04">]
```

As expected, one book matched. So far so good. Now let's say we want the results to be grouped into buckets, or *facets*. We just add a facet for the publisher field, for example:

```
class Book < ActiveRecord::Base
  acts_as_solr :facets => [:publisher]
  has_and_belongs_to_many :categories
end
```

Changing the acts_as_solr configuration may require reindexing, which we can do using the rebuild_solr_index method for the model that has changed. We can do that from the console, too:

```
>> reload!
>> Book.rebuild_solr_index
```

Then we run the same query, but this time telling find_by_solr that we're interested in how many results are in the publisher facet:

```
>> results = Book.find_by_solr("recipe", :facets => {:fields => [:publisher]})
=> #<ActsAsSolr::SearchResults:0x1a46950...>
>> results.total
=> 1
>> results.facets
=> {"facet_queries"=>{},
    "facet_fields"=>{"publisher_facet"=>{"See The Light Publishing"=>1}}}
```

We searched for "recipe". Out of all documents that match that constraint, how many are from each publisher? The facets method lets us see how many results fall into various buckets (or *facets*). This can help provide a richer browsing experience for users—the kind of drill down through categories that you might see on a shopping site. We could, for example, use Solr's filter query capability to further constrain results by selected facets.

Adding the book categories and published years as facets requires a few magic incantations, but we're feeling lucky. We need to change our Book model to this:

```
class Book < ActiveRecord::Base
  acts_as_solr :facets => [:publisher, :category, :year],
               :additional_fields => [:category, :year]

  has_and_belongs_to_many :categories

  def category
    categories.collect {|c| c.name }
  end

  def year
    published_date ? published_date.year : nil
  end
end
```

We've added a couple accessor methods—category and year—to bundle up lower-level data. The :additional_fields option gets acts_as_solr to pull in these "synthetic" fields. We also specify those same field names as facet fields, which causes them to be named and treated different by Solr.

Unless otherwise specified, all fields are assumed to be text fields (*_t in the Solr schema). Fields specified in the :facets array are named *_facet in the Solr documents. The acts_as_solr plugin expends great effort to map field names automatically between Active Record and Solr, but the Solr facet field names leak through acts_as_solr, as shown with the _facet suffixes.

Once again we need to run Book.rebuild_solr_index, and now we can peek at the new facets in the console:

```
>> reload!
>> Book.rebuild_solr_index
>> results = Book.find_by_solr("solr",
:facets => {:fields => [:category, :year]})
=> #<ActsAsSolr::SearchResults:0x17b883c...>
>> results.facets
=> {"facet_queries"=>{}, "facet_fields"=> {
       "category_facet"=>{"Information Retrieval"=>1, "Yummy"=>1},

    "year_facet"=>{"2007"=>1}}}
```

All that's left is putting it on the web. That's the easy part. Imagine we have a search action like this:

Download ActsAsSolr/app/controllers/books_controller.rb

```
def search
  @results = Book.find_by_solr(params[:term],
             :facets => {:fields => [:category, :year]})
end
```

Then we could render our search results like so:

Download ActsAsSolr/app/views/books/search.html.erb

```
<h1><%= pluralize(@results.total, 'result') %></h1>

<ul>
<% for book in @results.docs %>
  <li><%= link_to book.title, book %></li>
<% end %>
</ul>
```

And that's all there is to it! Finally, to stop the Solr server, we use:

```
$ rake solr:stop
```

When tinkering around, you may want to start with a fresh Solr index. Do do this, stop Solr, delete that directory, and restart Solr. The search index data resides in vendor/plugins/acts_as_solr/solr/solr/data by default.

So we've put together full-text searching *with* faceting in fairly short order, with very little changes to our Active Record models.

Discussion

There are a couple of additional bells and whistles in acts_as_solr worth mentioning:

- autocommit control, so batch indexing can commit to Solr at the end instead of for every record, and
- multi-model search, allowing full-text searches to span ActiveRecord models

Also See

If you fancy a Ruby alternative, see Recipe 7, *Full-Text Search with Ferret*, on page 40.

Part III

Database Recipes

Recipe 9

Adding Foreign Key Constraints

Problem

You want to add foreign key constraints to your database to, you know, ensure referential integrity. That way you can't accidentally delete records that other ones refer to, be it through your Rails application or another application that shares your database.

Solution

Let's say we have a classic order, line item, and product model arrangement. A line item points to both a product and an order.

Download buffet/app/models/line_item.rb

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :product
end
```

Download buffet/app/models/order.rb

```
class Order < ActiveRecord::Base
  has_many :line_items
end
```

Download buffet/app/models/product.rb

```
class Product < ActiveRecord::Base
  has_many :line_items
end
```

The only interesting migration file is for the line_items table:

Download buffet/db/migrate/006_create_line_items.rb

```
class CreateLineItems < ActiveRecord::Migration
  def self.up
    create_table :line_items do |t|
      t.integer :product_id, :null => false
      t.integer :order_id,   :null => false
    end
  end
  def self.down
    drop_table :line_items
  end
```

So to create a line item, we must have both an order and a product:

```
item = LineItem.create(:order => an_order, :product => a_tshirt)
```

In other words, it makes no sense to have a line item that doesn't reference a product. But then we can turn around and delete the a_tshirt record from the database, leaving the line item holding nil product. That's no good—let's fix it!

Databases are smart about keeping invariants like this in check. Migrations don't support adding foreign key constraints, but we can make it look as though they do. Here's the revised migration for line items:

Download buffet/db/migrate/006_create_line_items.rb

```
class CreateLineItems < ActiveRecord::Migration

  extend MigrationHelpers

  def self.up
    create_table :line_items do |t|
      t.integer :product_id, :null => false
      t.integer :order_id,   :null => false
    end
    fk :line_items, :product_id, :products
    fk :line_items, :order_id,   :orders
  end

  def self.down
    drop_fk :line_items, :order_id
    drop_fk :line_items, :product_id
    drop_table :line_items
  end
end
```

The fk method adds the constraint when the migration is applied, and the drop_fk method does the reverse. We've extended this migration class to support these methods, which are tucked away in the MigrationHelpers module. Here's what's in that module:

Download buffet/lib/migration_helpers.rb

```
module MigrationHelpers

  def fk(from_table, from_column, to_table)
    execute %(alter table #{from_table}
              add constraint #{constraint_name(from_table, from_column)}
              foreign key (#{from_column})
              references #{to_table}(id))
  end

  def drop_fk(from_table, from_column)
```

```
    execute %(alter table #{from_table}
              drop foreign key #{constraint_name(from_table, from_column)})
  end

  def constraint_name(table, column)
    "fk_#{table}_#{column}"
  end

end
```

Using the execute method lets us run arbitrary SQL inside a migration, and this module keeps it all in one tidy spot. Better yet, we can call these helpers from any migration file that extends the module.

So now if we try to delete a product that a line item is pointing to, we get an exception:

```
ActiveRecord::StatementInvalid: Mysql::Error: Cannot delete or update
a parent row: a foreign key constraint fails
(`buffet_development/line_items`, CONSTRAINT `fk_line_items_product_id`
FOREIGN KEY (`product_id`) REFERENCES `products` (`id`)):
DELETE FROM `products` WHERE `id` = 1
```

And that's exactly what we want to happen. When a foreign key constraint fails, we've broken a fundamental truth (an invariant) in our business logic. That calls for an explicit action on our part to deal with it.

Discussion

As of Rails 2.0, referential integrity checking is disabled while test fixtures are being created. That means you don't necessarily have to load fixtures in a specific order.

Also See

- The Foreign Key Migration plugin: http://www.redhillonrails.org/#foreign_key_migrations

Write Your Own Custom Validations

By Matthew Bass (http://matthewbass.com)
Matthew Bass is an independent software developer who has been enjoying the freedom of Ruby for many years now. He is a speaker, agile evangelist, and Mac addict. He co-organizes the Ruby Meetup in his home town of Raleigh, North Carolina and blogs at http://matthewbass.com.

Problem

Rails gives us a solid set of model validations right out of the box. But what if you need to write application-specific validations, and share them across models?

Solution

Say we have a Student model that validates the presence of a name and that the social security number is of the format ###-##-#### (where each pound sign is a number), like this:

```
class Student < ActiveRecord::Base
  validates_presence_of :name
  validates_format_of :ssn,
                      :with => /^[\d]{3}-[\d]{2}-[\d]{4}$/,
                      :message => "must be of format ###-##-####"
end
```

Then imagine we add a new Teacher model and teachers also need to have valid social security numbers. Now, we could copy/paste the SSN formatting, but as advanced programmers we know better. Instead, we'd like to write a declarative validation we can reuse, something like this:

Download CustomValidations/app/models/student.rb

```
class Student < ActiveRecord::Base
  validates_presence_of :name
  validates_ssn :ssn
end
```

Download CustomValidations/app/models/teacher.rb

```
class Teacher < ActiveRecord::Base
  validates_presence_of :name
  validates_ssn :ssn
```

```
end
```

And while we're asking for stuff, we might as well handle multiple SSN attributes, like so:

```
validates_ssn :lost_ssn, :replacement_ssn
```

OK, so how do we get there? Well, first we need to create a class-level method that can be invoked from any subclass of ActiveRecord::Base. We'll tuck away our call to validates_format_of in that class method, calling it once per attribute. Our class method, by itself, looks like this:

```
def self.validates_ssn(*attr_names)
  attr_names.each do |attr_name|
    validates_format_of attr_name,
                        :with => /^[\d]{3}-[\d]{2}-[\d]{4}$/,
                        :message => "must be of format ###-##-####"
  end
end
```

Now, how do we get this method into ActiveRecord::Base? It turns out we have several options. One way is to open up the ActiveRecord::Base class and define our class method inline:

```
class ActiveRecord::Base
  def self.validates_ssn(*attr_names)
    attr_names.each do |attr_name|
      validates_format_of attr_name,
                          :with => /^[\d]{3}-[\d]{2}-[\d]{4}$/,
                          :message => "must be of format ###-##-####"
    end
  end
end
```

This isn't a bad option per se, but it can be somewhat difficult to test under certain conditions. By sticking our class method in its own module and then extending that module, we accomplish the same goal without encumbering testability:

Download CustomValidations/lib/validations.rb

```
module CustomValidations

  def validates_ssn(*attr_names)
    attr_names.each do |attr_name|
      validates_format_of attr_name,
                          :with => /^[\d]{3}-[\d]{2}-[\d]{4}$/,
                          :message => "must be of format ###-##-####"
    end
  end

end
```

```
ActiveRecord::Base.extend(CustomValidations)
```

Here we're gently mixing our custom validation methods in to the ActiveRecord::Base class. That is, we use extend to add the methods of our CustomValidations module to the ActiveRecord::Base class object.

This is a better design because we can mix in our custom validation methods only when we need them. And it leads to better code organization. If we ever decide to write more validation methods, we can simply add them to the CustomValidations module.

To put our nifty new validations to use, we drop this code in a file called custom_validations.rb, for example, in the lib directory of our Rails application. Then we require it in our environment.rb file and we can start using the validates_ssn validation in all of our ActiveRecord models.

Finally, we need to test all this. We can sidestep the database completely by just calling valid?, and optionally checking the errors collection:

Download CustomValidations/test/unit/student_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class StudentTest < ActiveSupport::TestCase

  def test_validation_succeeds
    s = Student.new(:name => "Charlie Brown", :ssn => "123-12-1234")
    assert s.valid?
  end

  def test_validation_fails
    s = Student.new(:name => "Linus", :ssn => "1234")
    assert !s.valid?
    assert_equal "must be of format ###-##-####", s.errors[:ssn]
  end
end
```

Download CustomValidations/test/unit/teacher_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class TeacherTest < ActiveSupport::TestCase

  def test_valid
    t = Teacher.new(:name => 'Miss Othmar')
    t.ssn = "123-12-1234"
    assert t.valid?
  end

  def test_invalid
```

```
    t = Teacher.new(:name => 'Miss Othmar')
    t.ssn = "1234"
    assert !t.valid?
    assert_equal "must be of format ###-##-####", t.errors[:ssn]
  end
end
```

Encapsulating validations like this leads to far more readable model classes, not to mention that warm feeling you get when you realize your code is high and DRY.

Analyzing SQL Queries

By Pierre-Alexandre Meyer (http://www.mouraf.org)
Pierre-Alexandre is a 21 year old French application developer specializing in Ruby on Rails. He's currently pursuing a Master's Degree at Cornell University.

Problem

Analyzing your database queries is a fundamental task before deploying your application to the big, bold world. To do that, you could walk through the main pages of your app and watch the SQL that gets spewed out in the log file. But you already have integration tests for the well-worn paths, so wouldn't it be convenient if you could extract the SQL scenarios from those tests and use them with your favorite SQL benchmarking tools?

Solution

The first step is to collect the SQL generated by Active Record. Rails uses database-specific adapters to execute SQL operations. Thankfully, every adapter logs the SQL being run by calling the log_info method in the AbstractAdapter class. So we'll just intercept that call and stash away the SQL statements:

Download SqlLogging/config/initializers/core_extensions.rb

```
if RAILS_ENV == "test"
  class ActiveRecord::ConnectionAdapters::AbstractAdapter

    @@queries = []
    cattr_accessor :queries

    def log_info_with_trace(sql, name, runtime)
      return unless @logger and @logger.debug?
      self.queries << sql
      log_info_without_trace(sql, name, runtime)
    end

    alias_method_chain :log_info, :trace

  end
end
```

We've added behavior onto the log_info method using alias_method_chain. This is effectively the same as writing:

```
alias_method :log_info_without_trace, :log_info
alias_method :log_info, :log_info_with_trace
```

Whenever the log_info method is called, the log_info_with_trace method will be executed first and then the log_info_without_trace method (which is an alias for the original log_info method) is called.

Although we could capture all the queries in memory all the time, it's probably not wise in production. We just want to know which SQL statements were run during our integration tests. To do that, notice that we've wrapped the AbstractAdapter class definition in a condition so that we only hook into the log_info method when we're in test mode.

OK, let's see how this works in the console:

```
$ ruby script/console test
Loading test environment (Rails 2.0.1)
>> ActiveRecord::ConnectionAdapters::AbstractAdapter::queries
=> []
>> Order.find :first
=> #<Order id: 1...>
>> Order.find_by_name('Pierre-Alexandre')
=> #<Order id: 1...>
>> ActiveRecord::ConnectionAdapters::AbstractAdapter::queries
=> ["SET NAMES 'utf8'", "SET SQL_AUTO_IS_NULL=0", "SELECT * FROM `orders`
   LIMIT 1", "SHOW FIELDS FROM `orders`", "SELECT * FROM `orders`
   WHERE (`orders`.`name` = 'Pierre-Alexandre')  LIMIT 1"]
```

Great, we're capturing SQL behind Active Record's back. The next step is to modify our integration tests to siphon off the SQL statements into a file for our benchmarking tools to slurp up. To do that, we just wrap the test_create_order method in a trace_sql block. Here are the relevant parts:

Download SqlLogging/test/integration/story_test.rb

```
class StoryTest < ActionController::IntegrationTest

  def test_create_order
    trace_sql do
      go_to_orders
      place_order :name => 'Pierre-Alexandre', :total => 25.00
    end
  end

private

  def trace_sql
    yield
    File.open("#{RAILS_ROOT}/tmp/integration.sql", "w") do |file|
      queries = ActiveRecord::ConnectionAdapters::AbstractAdapter::queries.
```

```
                join("\n")
      file.write queries
    end
  end
end
```

Finally, after running our integration test, the SQL appears in the tmp/integration.sql file:

```
SET NAMES 'utf8'
SET SQL_AUTO_IS_NULL=0
BEGIN
SELECT * FROM `orders`
INSERT INTO `orders` (`city`, `name`, `updated_at`, `country`, `total`,
`created_at`) VALUES(NULL, NULL, '2007-12-14 07:22:45', NULL, NULL,
'2007-12-14 07:22:45')
```

Discussion

This technique of opening up Rails internal classes and hooking into methods (often called *monkey patching*) is powerful, and at the same time potentially dangerous. Future versions of Rails may change internal workings and break our code. In this case, log_info is a public method of a heavily-used API. It's unlikely that this method would change fundamentally, but it's always possible.

Also See

Two plugins in particular use a similar technique to give you insight into database activity:

- The query_trace plugin[11] dumps the stack trace of where your application is at when a SQL statement is run. It's a great plugin for pinpointing the exact location of a problematic query.
- The query_analyzer plugin[12] prints out the MySQL execution plan in your logs (using the MySQL EXPLAIN statement).

11. http://terralien.com/projects/querytrace/
12. http://svn.nfectio.us/plugins/query_analyzer

Recipe 12

Taking Advantage of Master/Slave Databases

By Rick Olson (http://activereload.net/)
Thanks to Rick Olson for technical help with this recipe.

Problem

Scalability is one of those fighting words. (In fact, if you really want to see how well your blog scales, just write a post stating that Rails can't possibly scale.) On a practical note, you have a number of knobs to turn and levers to pull to help your application scale.

One naive approach to improving scalability is to throw more Mongrel processes into the mix. But that's futile if your database is the real bottleneck. In that situation, after tuning your SQL queries, you may want to partition database access into read and write operations using master/slave database replication. Once you've set that up at the database level, how do you arrange things in your application to take advantage of it?

Ingredients

- Rick Olson's masochism plugin:

```
$ script/plugin install http://ar-code.svn.engineyard.com/plugins/masochism/
```

Solution

Here's the situation: We have an application that lets people give shouts out to the world, you know, something like Twitter. When someone shouts, we want it to go to the master (write) database. The database server will then take care of replicating the shout down to our slave database. Lots of people listen for shouts from their friends, so we want reading shout records to go through the slave database.

Once we've configured replication at the database server level[13], the masochism plugin lets us transparently take advantage of the master/slave

13. As database replication is very specific to your database, it's beyond the scope of this recipe. Refer to http://dev.mysql.com/doc/refman/5.0/en/replication-howto.html for MySQL instructions, for example.

database arrangement. First we need to configure the master and slave database in our config/database.yml file:

```
master_database:
  adapter: mysql
  database: the_master_database
  host: master.host.name
  ...

production:
  adapter: mysql
  database: the_slave_database
  host: slave.host.name
  ...
```

By default, the masochism plugin will use the database labeled master_database for write operations, and whatever database you have configured for your current environment as the read-only/slave database. In this case, if we're running in production the slave database will be the_slave_database. In production we'd likely set up the slave database on a different host than the master.

The masochism plugin makes all this transparent by swapping the database connections for us depending on the underlying Active Record operation. Update statements, operations in a transaction, and reloads are sent to the master database, and the rest go to the slave database. So next we need to set up the connection proxy by adding this to our config/environments/production.rb file:

```
config.after_initialize do
  ActiveReload::ConnectionProxy.setup!
end
```

OK, so how do we test this? One easy way is to try it without database replication being configured. If we write a record, it should only show up in the master database. Let's try that in the console:

```
$ ruby script/console -e production
>> Shout.create :name => "Mike", :shout => "Just ate some sushi!"
=> #<Shout id: 3...>
>> Shout.find(:all)
=> []
```

As expected, we gave a shout out and it was written to the master database. But since we haven't configured database replication between the master and slave, calling find doesn't find the shout. This tells us that the find is using the slave database. In other words, we know our Active Record operations are flowing to the appropriate database.

If we were now to configure master/slave replication at the database level, we should end up with shouts in the slave database:

```
>> Shout.find(:all)
=> [#<Shout id: 3 ...]
```

Now we can distribute the bulk of the database load across a number of slave databases, and let the master database focus on handling write operations.

Discussion

Using slave databases has some disadvantages, primarily the replication lag. While the replication is in progress, the slave database won't have the latest data. If you need to have the latest data in some parts of your code, you can make the finders fall back to the master database by wrapping the call in a transaction:

```
>> Shout.transaction { Shout.find(:all) }
=> [#<Shout id: 3 ...]
```

If you have a model that should use the master database for *all* operations, just change the model to subclass ActiveReload::MasterDatabase:

```
class SuperShout < ActiveReload::MasterDatabase
end
```

Part IV

User Interface Recipes

Replacing In-View Raw JavaScript with RJS

By Jared Haworth (http://www.alloycode.com/)
Jared Haworth is working to change the world as a Senior Software Engineer at Education Revolution. He is also actively involved in Rails Advocacy through his company Alloy Code, a North Carolina based web development shop.

Problem

You need a JavaScript-powered link or button. You're already using RJS to generate JavaScript in other places of your application, or you just don't care to write raw JavaScript. You want to use familiar RJS syntax to generate JavaScript for the link and button functions, too.

Solution

The link_to_function and button_to_function helpers will happily take any JavaScript as the function parameter. The often overlooked update_page method gives us a JavaScript generator (called page by convention) just like we'd get in an RJS file. In fact, there's nothing special about the page object in an RJS file.

So if we need to toggle the visibility of a quick_help div, for example, we'd use:

Download ReplacingRawJSWithRJS/app/views/events/new.html.erb

```
<%= link_to_function 'Help', update_page { |page| page[:quick_help].toggle } %>
<div id="quick_help" style="display: none;">
  Here's some help...
</div>
```

The update_page code block spits out Prototype-flavored JavaScript and substitutes it into the view at the time the page is rendered.

As a matter of practicality, the link_to_function and button_to_function helpers will also take a block if no function is specified. The block parameter is (you guessed it) a JavaScript generator. This makes it easy to update multiple elements or add special effects in a multi-line stanza:

```
<%= link_to_function('Help', nil) do |page|
  page[:quick_help].toggle
  page[:quick_help].visual_effect :highlight
```

```
end %>
```

Now we're moving in the right direction, but this inline code can get messy quickly. So let's take this a step further by refactoring our link_to_function calls to encourage reusability across views. View helpers are a good place to stash reusable view code, and as if by design we can call the update_page method from a helper. So in application_helper.rb we just bottle up the toggling code in a helper method:

Download ReplacingRawJSWithRJS/app/helpers/application_helper.rb

```
def toggle_div(div)
  update_page do |page|
    page[div].toggle
  end
end
```

Now we've got a helper that generates JavaScript to toggle the visibility of whatever DOM element we give it. Here's how we call it:

Download ReplacingRawJSWithRJS/app/views/events/show.html.erb

```
<p>
  <%= link_to_function "Show Venue Details", toggle_div(:venue_details) %>
</p>
<div id="venue_details" style="display: none;">
  <%= render :partial => 'venue', :object => @event.venue %>
</div>
```

The upshot of moving this code around until it found a comfy home is now we can toggle things from any view without the DRY police crying foul:

Download ReplacingRawJSWithRJS/app/views/venues/show.html.erb

```
<p>
  <%= link_to_function "Show Event Details", toggle_div(:event_details) %>
</p>
<div id="event_details" style="display: none;">
        <%= render :partial => "event", :collection => @venue.events %>
</div>
```

Where it makes sense, we can also call these helpers from controllers in response to a full AJAX request.

Give your helper methods good names and your views start to become quite readable, not to mention easier to maintain.

Handling Multiple Models In One Form

By Ryan Bates (http://railscasts.com/)
Ryan Bates has been involved in web development since 1998. In 2005 he started working professionally with Ruby on Rails and is now best known for his work on Railscasts, the free Ruby on Rails screencast series.

Problem

Most of the form code you see handles one model at a time. That's not always practical. Sometimes you need to create and/or update two (or more) models in a single form, where there is a one-to-many association between them.

Solution

Let's say we're keeping track of tasks we need to do on projects. When we create or update a project, we'd like to add, remove, and update its tasks in a single form.

Let's start with a has_many relationship between Project and Task. To keep things simple, we'll give each model a required attribute called name.

```
class Project < ActiveRecord::Base
  has_many :tasks, :dependent => :destroy
  validates_presence_of :name
end
```

```
class Task < ActiveRecord::Base
  belongs_to :project
  validates_presence_of :name
end
```

We'll be using the Prototype JavaScript library, so before we go any further let's make sure it's loaded in our layout file:

Download MultiModelForm/app/views/layouts/application.html.erb

```
<%= javascript_include_tag :defaults %>
```

Now we turn our attention to the form for creating a project along with its associated, multiple tasks.

When dealing with multiple models in one form it's helpful to make one model the primary focus and build the other models through the association. In this case, we'll make the Project model the primary and build its tasks through the has_many association. So in the new action of our ProjectsController we create a Project object like normal, but we also add three tasks to the project (in memory) so that our form has something to work with:

Download MultiModelForm/app/controllers/projects_controller.rb

```
def new
  @project = Project.new
  3.times { @project.tasks.build }
end
```

The form template is a bit tricky since we need to handle fields for the Project model and each of its Task models. So let's break the problem down a bit by using a partial to render the Task fields and a helper to create the link that adds a new task:

Download MultiModelForm/app/views/projects/_form.html.erb

```
<%= error_messages_for :project %>

<% form_for @project do |f| -%>
  <p>
    Name: <%= f.text_field :name %>
  </p>
  <div id="tasks">
    <%= render :partial => 'task', :collection => @project.tasks %>
  </div>
  <p>
    <%= add_task_link "Add a task" %>
  </p>
  <p>
    <%= f.submit "Submit" %>
  </p>
```

```
<% end -%>
```

Before we get into the contents of the task partial, let's take a look at that add_task_link helper method:

Download MultiModelForm/app/helpers/projects_helper.rb

```
def add_task_link(name)
  link_to_function name do |page|
    page.insert_html :bottom, :tasks, :partial => 'task', :object => Task.new
  end
end
```

When we click the "Add a task" link, we want a new set of task fields to appear at the bottom of the existing task fields in the form. Rather than bother the server with this, we can use JavaScript to add the fields dynamically. The link_to_function method accepts a block of RJS code. We usually associate RJS code with asynchronous calls back to the server. But in this case the RJS code generates JavaScript that gets executed in the browser immediately when the user clicks the link. The upshot is rendering the fields for adding a new task does not require a trip back to the server, which leads to faster response times.

Looking back to the form partial, we're using form_for to dedicate the form to the @project model. How then do we add fields for each of the project's tasks? The task partial holds the answer:

Download MultiModelForm/app/views/projects/_task.html.erb

```
<div class="task">
<% new_or_existing = task.new_record? ? 'new' : 'existing' %>
<% prefix = "project[#{new_or_existing}_task_attributes][]" %>

<% fields_for prefix, task do |task_form| -%>
  <p>
    Task: <%= task_form.text_field :name %>
    <%= link_to_function "remove", "$(this).up('.task').remove()" %>
  </p>
<% end -%>
</div>
```

The key ingredient here is the fields_for method. It behaves much like form_for, but does not render the surrounding form HTML tag. This lets us switch the context to a different model in the middle of a form—as if we're embedding one form within another.

The first parameter to fields_for is very important. This string will be used as the prefix for the name of each task form field. As we'll be using this partial to also render existing tasks—and we want to keep

them separate when the form is submitted—in the prefix we include an indication of whether the task is new or existing. (Ideally we'd create the prefix string in a helper, but we've inlined it here to avoid further indirection.)

The generated HTML for a new task name input looks like this:

```
<input name="project[new_task_attributes][][name]" size="30" type="text"/>
```

If this were an existing task, Rails would automatically place the task id between the square brackets, like this:

```
<input name="project[existing_task_attributes][7][name]" size="30" type="text"/>
```

Now when the form is submitted, Rails will decode the input element's name to impose some structure in the params hash. Square brackets that are filled in become keys in a nested hash. Square brackets that are empty become an array. For example, if we submit the form with three new tasks, the params hash looks like this:

```
"project" => {
  "name" => "Yard Work",
  "new_task_attributes" => [
    { "name" => "rake the leaves" },
    { "name" => "paint the fence" },
    { "name" => "clean the gutters" }
  ]
}
```

Notice that the attributes for the project *and* each task are nestled inside the project hash. This is convenient because it means that the create action back in our controller can simply pass all the project attributes through to the Project model without worrying about what's inside:

Download MultiModelForm/app/controllers/projects_controller.rb

```
def create
  @project = Project.new(params[:project])
  if @project.save
    flash[:notice] = "Successfully created project and tasks."
    redirect_to projects_path
  else
    render :action => 'new'
  end
end
```

This looks like a standard create action for a single-model form. There's just one problem: When we call Project.new(params[:project]) Active Record assumes that our Project model has a corresponding attribute called

new_task_attributes because it sees a key called new_task_attributes in the params(:project) hash. That is, Active Record will try to mass assign all the data in the params(:project) hash to corresponding attributes in the Project model.

One convenient way to keep all this transparent from the controller's perspective is to use a virtual attribute. To do that, we just create a setter method in our Project model called new_task_attributes= which takes an array and builds a task for each element:

Download MultiModelForm/app/models/project.rb

```
def new_task_attributes=(task_attributes)
  task_attributes.each do |attributes|
    tasks.build(attributes)
  end
end
```

It may not look like these tasks are being saved anywhere, but Rails will do that automatically for us once the project is saved because both the project and its associated tasks are new records.

That's it for creating a project, now let's move on to updating one.

Just like before, we need to be able to add and remove tasks dynamically, but this time if a task already exists it should be updated instead. The controller actions only need to be concerned about the project so they are fairly conventional. As before, the updating of the tasks will be handled in the Project model:

Download MultiModelForm/app/controllers/projects_controller.rb

```
def edit
  @project = Project.find(params[:id])
end

def update
  @project = Project.find(params[:id])
  if @project.update_attributes(params[:project])
    flash[:notice] = "Successfully updated project and tasks."
    redirect_to project_path(@project)
  else
    render :action => 'edit'
  end
end
```

The form partial can stay the same. However, when we submit the form with existing tasks, the params(:project) hash will include a key called existing_task_attributes. So we need to add an existing_task_attributes=

method to our Project model which will take each existing task and either update it or destroy it depending on if the attributes are passed:

Download MultiModelForm/app/models/project.rb

```
after_update :save_tasks

def existing_task_attributes=(task_attributes)
  tasks.reject(&:new_record?).each do |task|
    attributes = task_attributes[task.id.to_s]
    if attributes
      task.attributes = attributes
    else
      tasks.delete(task)
    end
  end
end

def save_tasks
  tasks.each do |task|
    task.save(false)
  end
end
```

Notice that we're saving the tasks in an after_update callback. This is important because unlike before, the existing tasks will not automatically be saved when the project is updated.[14] And since callbacks are wrapped in a transaction, it will properly roll back the save if an unexpected problem occurs.

Passing false to the task.save method bypasses validation. Instead, to ensure that all the tasks get validated when the project is validated, we just add this line to the Project model:

```
validates_associated :tasks
```

This ensures everything is valid before saving. And if validation fails, then the use of error_messages_for :project in the form template includes the validation errors for the project and any of its tasks.

So now we can create and edit projects and their tasks in one fell swoop. And by using virtual attributes, we kept the controller code happily ignorant that we were handling multiple models from a single form.

14. This behavior can vary depending on the type of association and whether the records are new. It's a good idea to thoroughly test each combination to ensure every model is validated and saved properly.

Also See

For an alternative approach, see Recipe 15, *Simplifying Controllers With a Presenter*, on the next page.

Recipe 15

Simplifying Controllers With a Presenter

By Jay Fields (http://jayfields.com)
Jay Fields is a software developer and consultant at ThoughtWorks. He has a passion for discovering and maturing innovative solutions. His most recent work has been delivering large enterprise applications utilizing Ruby and Rails. He is also very interested in maturing software design through software testing.

Problem

As your application has grown, so have your controllers. Rather than just orchestrating the work, they've taken on the responsibility of aggregating data from various objects to make life simpler for the views. As a result, maintainability has been compromised. You need to breathe new life into the controllers.

Solution

To illustrate the problem, imagine we have a controller that's responsible for creating, populating, and saving three models:

Download PresenterPattern/app/controllers/orders_controller.rb

```
class OrdersController < ApplicationController

  def new
    @account    = UserAccount.new
    @address    = Address.new
    @credential = UserCredential.new
  end

  def create
    @account    = UserAccount.new(params[:account])
    @address    = Address.new(params[:address])
    @credential = UserCredential.new(params[:credential])

    account_saved = @account.save
    @address.user_account = @account
    @credential.user_account = @account
    if account_saved && @address.save && @credential.save
      redirect_to thank_you_url
    end
  end
```

```
end
```

And to collect all the data in one form, we have the following new.html.erb template:

```
<% form_tag(:action => 'create') do -%>
  <table>
    <tr><td colspan="2">Account Information:</td></tr>
    <tr>
      <td>Name</td>
      <td><%= text_field :account, :name %></td>
    </tr>
    <tr><td colspan="2">Address Information:</td></tr>
    <tr>
      <td>Address Line 1</td>
      <td><%= text_field :address, :line_1 %></td>
    </tr>
    <tr>
      <td>Address Line 2</td>
      <td><%= text_field :address, :line_2 %></td>
    </tr>
    <tr>
      <td>City</td>
      <td><%= text_field :address, :city %></td>
    </tr>
    <tr>
      <td>State</td>
      <td><%= text_field :address, :state %></td>
    </tr>
    <tr>
      <td>Zip Code</td>
      <td><%= text_field :address, :zip_code %></td>
    </tr>
    <tr><td colspan="2">Credential Information:</td></tr>
    <tr>
      <td>Username</td>
      <td><%= text_field :credential, :username %></td>
    </tr>
    <tr>
      <td>Password</td>
      <td><%= text_field :credential, :password %></td>
    </tr>
  </table>
  <%= submit_tag "Complete Order" %>
<% end -%>
```

This works, but the controller is an eyesore. Also, testing individual behaviors, such as that the redirect does not occur if the credential doesn't save correctly, is a bit of a pain.

The solution is to introduce an intermediate object—a *presenter*—to

relieve some burden from the controller, while at the same time keeping the view simple. First, we'll change the controller to be more concise and focused:

Download PresenterPattern/app/controllers/orders_controller.rb

```
def new
  @presenter = OrderPresenter.new(params[:presenter])
end

def create
  @presenter = OrderPresenter.new(params[:presenter])
  if @presenter.save
    redirect_to thank_you_url
  end
end
```

Now that we have only one instance variable being set in the controller, we change the view to use form_for, *and* change each field's name to include that field's model name:

Download PresenterPattern/app/views/orders/new.html.erb

```
<% form_for :presenter,
            :url => {:action => 'create'} do |form| %>
  <table>
    <tr><td colspan="2">Account Information:</td></tr>
    <tr>
      <td>Name</td>
      <td><%= form.text_field :account_name %></td>
    </tr>
    <tr><td colspan="2">Address Information:</td></tr>
    <tr>
      <td>Address Line 1</td>
      <td><%= form.text_field :address_line_1 %></td>
    </tr>
    <tr>
      <td>Address Line 2</td>
      <td><%= form.text_field :address_line_2 %></td>
    </tr>
    <tr>
      <td>City</td>
      <td><%= form.text_field :address_city %></td>
    </tr>
    <tr>
      <td>State</td>
      <td><%= form.text_field :address_state %></td>
    </tr>
    <tr>
      <td>Zip Code</td>
      <td><%= form.text_field :address_zip_code %></td>
    </tr>
```

```
    <tr><td colspan="2">Credential Information:</td></tr>
    <tr>
      <td>Username</td>
      <td><%= form.text_field :credential_username %></td>
    </tr>
    <tr>
      <td>Password</td>
      <td><%= form.text_field :credential_password %></td>
    </tr>
  </table>
  <%= form.submit "Complete Order" %>
<% end %>
```

The presenter itself is just a plain ol' Ruby object that encapsulates access to three models. Here's the code for it:

Download PresenterPattern/app/presenters/order_presenter.rb

```
class OrderPresenter

  def initialize(params)
    params.each_pair do |attribute, value|
      self.send :"#{attribute}=", value
    end unless params.nil?
  end

  def account
    @account ||= UserAccount.new
  end

  def address
    @address ||= Address.new
  end

  def credential
    @credential ||= UserCredential.new
  end

  def save
    account_saved = account.save
    address.user_account = account
    credential.user_account = account
    account_saved && address.save && credential.save
  end

  def method_missing(model_attribute, *args)
    model, *method_name = model_attribute.to_s.split("_")
    super unless self.respond_to? model.to_sym
    self.send(model.to_sym).send(method_name.join("_").to_sym, *args)
  end

end
```

There's an interesting trick here. When we pluck the form parameters out of the presenter hash, the parameter keys will have a model name and a corresponding attribute name. So, for example, the value of the account name will be indexed by the account_name key. We need to unravel that so the value is assigned to the name attribute of the account object living inside the presenter. To do that, we use method_missing to first intercept the call to account_name=, for example, and then forward it on to the account object. It keeps the presenter flexible, but it relies on careful naming of the form fields.

One last step and we're home free. We've added the presenter to the app/presenters directory, which Rails doesn't know about. So finally we need to add this directory to the Rails load path in environment.rb:

Download PresenterPattern/config/environment.rb

```
config.load_paths += %W( #{RAILS_ROOT}/app/presenters )
```

So we've lightened the load on our controller, and gained a presenter object that aggregates view data and can be tested without any dependencies on Rails.

Discussion

Arguably we should push the logic of the OrderPresenter.save method back into one of the models, preferably wrapped in a transaction.

Validating Required Form Fields Inline

By Jarkko Laine (http://jlaine.net)
Jarkko Laine is one of the earliest Rails evangelists in Europe, with more than two years of experience in teaching and giving talks about Rails. He wrote *Beginning Ruby on Rails E-Commerce: From Novice to Professional* (HL06) with Christian Hellsten and is the founder of the Finnish Rails user community. He currently works as a senior developer for http://dotherightthing.com, a site for rating and discussing the social performance of world's businesses. In his freetime, Jarkko runs through forests like a gnu and writes about anything he finds interesting on his weblog.

Problem

Rails has powerful and easy-to-use form validation mechanisms on the server side. However, from the user interface perspective it would be valuable to catch at least the most obvious input errors—such as missing elements that are required—before the form is even submitted.

Ingredients

- Michael Schuerig's validation_reflection plugin:

```
$ script/plugin install ↩
    svn://rubyforge.org//var/svn/valirefl/validation_reflection/trunk
```

Solution

If we're signing up new users, we need their e-mail and a password. So let's automatically mark those form fields as being required and validate the required fields actually contain values *before* the user presses the "Submit" button.

We'll tackle the solution in two steps. First, we'll extract validation information from a Rails model. Then we'll write a custom form builder (see Recipe 22, *Keeping Forms Dry and Flexible*, on page 115) to automatically decorate the required fields and validate them inline.

Models carry around a lot of useful information. For example, our User model knows that the email and password fields are mandatory because we said so:

```
class User < ActiveRecord::Base
  validates_presence_of :email, :password
end
```

The validation_reflection plugin lets us tease this information out. For example, we can call the reflect_on_validations_for method to see the validations for a specific field of our User model:

```
$ ruby script/console
>> User.reflect_on_validations_for(:email)
=> [#<ActiveRecord::Reflection::MacroReflection:0x1834a40
   @macro=:validates_presence_of, @name=:email, @options=nil,
   @active_record=User(id: integer, login: string, email: string,
   password: string, created_at: datetime, updated_at: datetime)>]

>> User.reflect_on_validations_for(:password)
=> [#<ActiveRecord::Reflection::MacroReflection:0x1834464
   @macro=:validates_presence_of, @name=:password, @options=nil,
   @active_record=User(id: integer, login: string, email: string,
   password: string, created_at: datetime, updated_at: datetime)>]
```

Now we can use this knowledge in our custom form builder:

Download InlineFormValidations/lib/validating_form_builder.rb

```
class ValidatingFormBuilder < ActionView::Helpers::FormBuilder

  helpers = field_helpers +
            %w(date_select datetime_select time_select) -
            %w(hidden_field label fields_for)

  helpers.each do |name|
   define_method(name) do |field, *args|
     options = args.last.is_a?(Hash) ? args.pop : {}

     @template.content_tag(:p,
                           label(field, label_text(field)) + " " +
                           super(field, options))
   end
  end

private

  def field_name(field)
   "#{@object_name.to_s.underscore}_#{field.to_s.underscore}"
  end

  def label_text(field)
    "#{field.to_s.humanize}#{required_mark(field)}"
  end

  def required_mark(field)
```

```
    required_field?(field) ? ' (*)' : ''
  end

  def required_field?(field)
    @object_name.to_s.camelize.constantize.
                 reflect_on_validations_for(field).
                 map(&:macro).include?(:validates_presence_of)
  end
end
```

At a first glance, this form builder looks just like a normal form builder. However, we've added the required_field? method to check whether a given field is required and if so we include a simple indicator (an asterisk) in the field's label text.

While this is very nifty, it's not very interactive. However, with a small modification and a bit of JavaScript we can make the form tell the user whether she missed a required field. First we add the following JavaScript to our application.js file (and remember to include it in our layout file):

Download InlineFormValidations/public/javascripts/application.js

```
function checkPresence(field) {
 var hint = $F(field).length == 0 ? "Try again!" : "Right on!";
 if ($(field + '_hint')) {
   $(field + '_hint').update(hint);
 } else {
   content = '<span class="validation" id="' + field + '_hint">' +
             hint + '</span>';
   new Insertion.After(field, content);
 }
}
```

This function checks whether a field is empty and inserts an appropriate message after the form field. Next we just need to call this function for our required form fields. We can do that back in our form builder by adding a check before the form field is generated:

Download InlineFormValidations/lib/validating_form_builder.rb

```
if %w(text_field password_field).include?(name) && required_field?(field)
  options[:onblur] = "checkPresence('#{field_name(field)}')"
end
@template.content_tag(:p,
                      label(field, label_text(field)) + " " +
                      super(field, options))
```

This code checks whether the helper we're creating is a required text or password field. If so, we add an onblur event handler that calls our

checkPresence JavaScript function whenever the focus is moved away from the form field.[15] Now our form output as seen by a browser looks something like this:

```
<form action="/users" method="post">
   <p>
     <label for="user_login">Login</label>
     <input id="user_login" name="user[login]" size="30" type="text" />
   </p>
   <p>
     <label for="user_email">Email (*)</label>
     <input id="user_email" name="user[email]"
            onblur="checkPresence('user_email')" size="30" type="text" />
   </p>
   <p>
     <label for="user_password">Password (*)</label>
     <input id="user_password" name="user[password]"
            onblur="checkPresence('user_password')" size="30" type="text" />
   </p>
   <p>
     <input name="commit" type="submit" value="Create" />
   </p>
 </form>
```

The email and password fields include asterisks in their labels, and when the user tabs through those fields (for example) the phrase “Right on!” or “Try again!” appears below the input field. And because this functionality is tucked away in our form builder, it applies to all forms that use the builder.

Discussion

You could easily extend the validation to also cater to more advanced validation such as text length or format (for example, for e-mail addresses). This way you could easily make your forms more user-friendly and at the same time save a few request cycles down to the server. You could also extend the form builder so that it would make Ajax requests to check whether a given unique login name is already taken, for example.

15. We use an inline event handler here instead of less obtrusive methods for the sake of brevity.

Creating a Wizard

By Mike Hagedorn (http://www.silverchairsolutions.com)
Mike Hagedorn is a freelance web developer and founder of Silverchair Solutions, an agile methods consulting firm located in Houston, TX. Mike has a long history of enterprise development dating back to the bad old days of Java 1.0 and has been actively doing work in Rails since late 2005. He has implemented solutions in Java, Cocoa (Objective-C) and C#. To round things out he also moonlights as a professional musician and spends as many weekends as possible backpacking.

Problem

You've used a Wizard before; it's those series of screens with "Previous" and "Next" buttons at the bottom. You can go backward and forward as many times as you like until you get things just right. Indeed, a good Wizard takes a user by the hand and guides her through a step-by-step process.

Unfortunately, Wizards aren't trivial to implement in web-based applications because the web is a stateless world. And in order to know which step you're on in a multi-step process, state is the very thing that you need most. So just how do you roll your own Wizard?

Ingredients

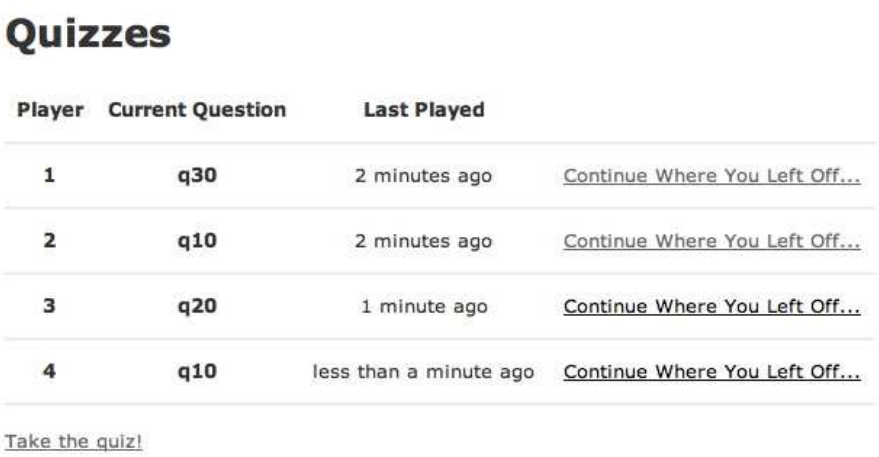
- Scott Barron's acts_as_state_machine plugin:

```
$ script/plugin install ↩
    http://elitists.textdriven.com/svn/plugins/acts_as_state_machine/trunk
```

Solution

Let's say we want to create an application that lets users take a short 3-question quiz. We'll need to keep track of which step of the quiz a user is currently on, and therefore which question to present to the user next (or previous). Here's an example question:

Of course lots of people will take our quizzes, and sometimes they'll need to step away to ponder a particularly tough question, but we'll let them pick up right where they left off:



Now that we know what we want, let's start with the models and migrations we'll need. A Quiz has many Answers and Questions, and it remembers its current state:

Download Wizard/app/models/quiz.rb

```
class Quiz < ActiveRecord::Base
  has_many :answers, :dependent => :destroy
  has_many :questions
end
```

Download Wizard/db/migrate/001_create_quizzes.rb

```
create_table :quizzes do |t|
  t.string :state
  t.timestamps
end
```

And a Question has many Answers—one for each person who took the quiz—plus the question text and some meta-data we'll get to later:

Download Wizard/app/models/question.rb

```
class Question < ActiveRecord::Base
  belongs_to :quiz
  has_many :answers
end
```

Download Wizard/db/migrate/003_create_questions.rb

```
create_table :questions do |t|
  t.string  :type, :text, :tag
  t.integer :quiz_id
```

```
  t.timestamps
end
```

And finally an Answer reciprocates the relationships and has a value for the answer:

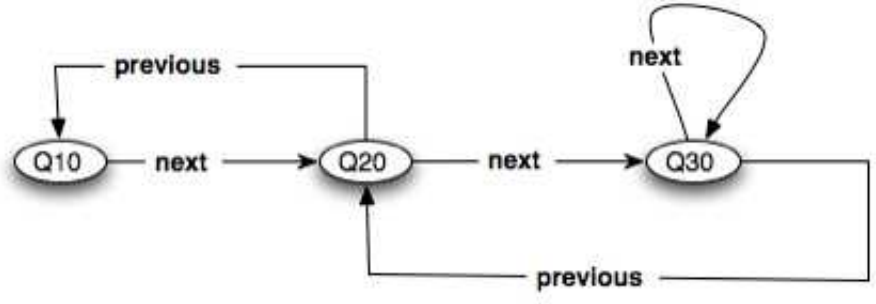
Download Wizard/app/models/answer.rb

```
class Answer < ActiveRecord::Base
  belongs_to :question
  belongs_to :quiz
end
```

Download Wizard/db/migrate/002_create_answers.rb

```
create_table :answers do |t|
  t.string   :value
  t.integer  :question_id, :quiz_id
  t.timestamps
end
```

That's fairly straightforward. Now, how are we going to tie all the questions together so that they get asked in a certain order? Well, we could use foreign keys in the database to point to the next and previous questions. But managing all that can get complicated as our quizzes become more involved. Thankfully, there's a simpler, more elegant way: a Finite State Machine (FSM). Using an FSM lets us break down the problem into a small number of *states* (the current question), and then move between the states when an event is encountered (the next or previous button is pressed). The mere thought of using a Finite State Machine sounds intimidating (and may trigger flashbacks to a time when you thought real programmers would never use them), but they're really easy. Let's look at ours graphically:



A quiz has three states with each state corresponding to a question to be posed. This small diagram completely defines what our 3-question quiz application should do when a *next* or *previous* event occurs:

- If you're in the first state (Q10) and you receive a next event, then transition to the next question (Q20).
- If you're in the second-question state (Q20) and you receive a next event, then transition to the third-question state (Q30). However, if you receive a previous event while in state Q20, go back to the first question (Q10).
- If you're in the third-question state (Q30) and receive a next event, do nothing. But if you receive a previous event, back up to question Q20.

That makes sense conceptually, but now how do we write code to make it work? Ah, that's where the acts_as_state_machine plugin comes in. We can express the states and transitions right in our Quiz model, like so:

Download Wizard/app/models/quiz.rb

```
class Quiz < ActiveRecord::Base
  has_many :answers, :dependent => :destroy
  has_many :questions

  acts_as_state_machine :initial => :q10

  state :q10, :after => :current_question
  state :q20, :after => :current_question
  state :q30, :after => :current_question

  event :next do
    transitions :to => :q20, :from => :q10
    transitions :to => :q30, :from => :q20
  end

  event :previous do
    transitions :to => :q10, :from => :q20
    transitions :to => :q20, :from => :q30
  end

  def current_question
    @current_question ||= find_question(self.current_state)
  end

private

  def find_question(state)
    Question.find_by_tag(state.to_s)
  end

end
```

We've added the acts_as_state_machine declaration, and set the initial state to q10 (which is just a name). Then we define the names of our three states and associate a method to call after transitioning into that state. In this case, the current_question method will turn around and query the database for the Question that has the same tag as the name of the current state. This is the key that makes the correct question pop up whenever a user hits the "Next" or "Previous" button. Then for each event (next and previous), we describe the transitions based on the current state.

Let's play around in the console a bit to get a feel for what's going on:

```
$ ruby script/console
>> quiz = Quiz.create
```

Notice that we didn't call the new method on Quiz. We have to use the create method in order for the acts_as_state_machine magic to kick in and set the initial state properly. Let's check that:

```
>> quiz.state
=> "q10"
```

Ok so far; that is indeed our first state. Let's move to the next state (which remember is the same thing as stepping to the next question). To do that, we can use the next! method because we defined next as an event:

```
>> quiz.next!
=> true
>> quiz.state
=> "q20"
```

Great, now let's go backwards using the previous! method:

```
>> quiz.previous!
=> true
>> quiz.state
=> "q10"
```

Then if we fast forward to the end, we loop back on the last state:

```
>> quiz.next!
>> quiz.next!
>> quiz.next!
>> quiz.state
=> "q30"
```

Hey, that's pretty neat! Next we need some questions. Quizzes are more fun if we have different types of questions: short answers, true or false, etc. Each of those requires a custom template to prompt the

user accordingly. To keep things flexible, we'll use single-table inheritance (STI) on the Question model. Then we can have each question type provide its own template for viewing purposes, and the rest of the system is none the wiser. We'll just have short answer and true/false questions for this quiz:

```
class ShortAnswerQuestion < Question
end

class TrueFalseQuestion < Question
end
```

Since we defined the Quiz class with states q10, q20 and q30, we need to create records in the questions table that have these tag values. Let's do that from the console, too:

```
$ ruby script/console
>> ShortAnswerQuestion.create(:text => "What's your name?",
                              :tag  => "q10")
>> TrueFalseQuestion.create(:text => "Do you like wasabi?",
                            :tag  => "q20").
>> ShortAnswerQuestion.create(:text => "What do you like to drink?",
                              :tag  => "q30")
```

Then over in a view helper we'll figure out which question template to use when showing a particular question. To keep things simple, we'll just use the convention of naming the template (we'll use partials) based on the class name of the current question:

Download Wizard/app/helpers/quizzes_helper.rb

```
module QuizzesHelper
  def question_template(question)
    "questions/#{question.class.name.underscore}"
  end
end
```

Let's go ahead and create the two partials that correspond to our two question types. These need to go in the app/views/questions directory according to our naming convention. Here's what the partial for short answer questions looks like:

Download Wizard/app/views/questions/_short_answer_question.html.erb

```
<label for="answer_value"><%= question.text %></label>
<%= text_field :answer, :value %>
```

And here's the true/false question partial:

Download Wizard/app/views/questions/_true_false_question.html.erb

```
<label for="answer_value"><%= question.text %></label>
```

```
<%= select :answer, :value,
           {"Yes" => "true", "No" => "false"},
           :selected => answer.value %>
```

The last piece to tie this all together is the QuizzesController, where all the interesting stuff happens. Let's generate it with four actions:

```
$ script/generate controller quizzes index new edit update
```

We'll go with the RESTful flow in our forms, so we need to add this to our config/routes.rb file:

```
map.resources :quizzes
```

We only need to be concerned with two actions: edit and update. The edit action lets a user continue an existing quiz:

Download Wizard/app/controllers/quizzes_controller.rb

```
def edit
  @quiz   = Quiz.find(params[:id])
  @answer = @quiz.answers.
              find_by_question_id(@quiz.current_question.id) || Answer.new
end
```

The template for the edit action renders the partial for the current question, handing it the question and the previous answer:

Download Wizard/app/views/quizzes/edit.html.erb

```
<% form_for(@quiz) do |f| -%>
  <fieldset>
    <%= render :partial => question_template(@quiz.current_question),
               :locals => {
                 :question => @quiz.current_question,
                 :answer   => @answer
               } %>
  </fieldset>
  <%= hidden_field_tag :direction, "next!" %>
  <hr/>
  <table class="controls">
    <tr>
      <td>
        <%= button_to "&#8592; Previous",
              {:id => @quiz, :action => "update"},
              {:method => :put,
               :onclick => "$('direction').value = 'previous!';"} %>
      </td>
      <td>
        <%= submit_tag "Next &#8594;" %>
      </td>
    </tr>
  </table>
```

```
<% end -%>
```

We use a hidden field here to indicate which direction we're going. Hitting the "Next" button just posts to the update action which calls the Quiz#next! method. The "Previous" button works slightly different: before posting to update it does a switcheroo on the hidden field value so we'll end up transitioning back one state. Here's the update action:

Download Wizard/app/controllers/quizzes_controller.rb

```
def update
  @quiz = Quiz.find(params[:id])
  @answer = @quiz.answers.find_by_question_id(@quiz.current_question)

  if @answer
    @answer.update_attribute(:value, params[:answer][:value])
  else
    @answer = Answer.new(:value    => params[:answer][:value],
                         :question => @quiz.current_question)
    @quiz.answers << @answer
  end

  @quiz.send(params[:direction])

  redirect_to :action => :edit
end
```

Every time a user hits the "Previous" or "Next" button, they'll transparently cycle through the update action. Their answer gets updated (if they've posted an answer before) or gets created (if they haven't given an answer yet). Then an event is fired—next! or previous! is called on the quiz—and after the transition the quiz loads up the corresponding question. Finally the action re-renders the edit template to show the next question.

Now all we have to do is fire up our browser, create a new quiz, and start stepping through the wizard!

Discussion

OK, but what if you're not building an online quiz game? Instead, you have something with a tad more states and transitions. All the more reason to find the back of a napkin, draw your state transition diagram, and start implementing it with acts_as_state_machine. Then walk through your diagram using the console, and when you have it working write some unit tests that give you automated examples. By breaking it

down this way, the final solution will likely be easier (and more elegant) then you may have imagined.

It would be fairly easy to extend this example to handle other question types, such as multiple choice questions, by defining new Question subclasses and including a corresponding template in the questions directory. You could also generalize the question template that's used by adding a view_template attribute to Question and allowing that field value to override the default template name.

Updating Partial Resources with Ajax

By David Heinemeier Hansson (http://loudthinking.com)
Thanks to David Heinemeier Hansson for the idea for this recipe.

Problem

You want to update certain attributes of a resource with Ajax, such as toggling the value of one field. How do you handle it in a RESTful way?

Solution

Our dear readers often find mistakes or have suggestions, so let's say we have an interface for submitting a book erratum.

If an author is logged in, he sees a checkbox for marking each erratum as being fixed. (It's good fun, and a great relief from writer's block.) When he checks off an erratum, we want to update the Errata resource's fixed attribute. Rather than creating a new controller action just for this case, we can piggy-back on the standard update action.

Here's the check box part of each row in the list of errata:

Download AjaxRestToggle/app/views/errata/index.html.erb

```
<%= check_box_tag 'erratum[fixed]', "1", erratum.fixed,
                  :onclick => toggle_value(erratum) %>
<%= image_tag 'spinner.gif', :id => "spinner-#{erratum.id}",
                             :style => 'display: none' %>
```

When the check box is clicked, it calls the toggle_value helper method:

Download AjaxRestToggle/app/helpers/application_helper.rb

```
def toggle_value(object)
  remote_function(:url => url_for(object),
                  :method => :put,
                  :before => "Element.show('spinner-#{object.id}')",
                  :complete => "Element.hide('spinner-#{object.id}')",
                  :with => "this.name + '=' + this.checked")
end
```

This code uses remote_function to fire off an asynchronous request to the update action of the given resource. While that's happening we show the spinner to let the tireless author know that something is happenening. In this case the resource is an Erratum object, but the helper is generic—it will take any resource and use the check box name in the enclosing template. Here's what the actual HTTP request looks like for an erratum, for example:

PUT/errata/4?erratum(fixed)=1

It's a purely RESTful URL, and the advantage is that updating a single attribute (fixed) just piggybacks onto the full update action. So all we need to do to support the single-attribute update is add a new format.js block to respond_to to let the JavaScript request know that the update was successful:

Download AjaxRestToggle/app/controllers/errata_controller.rb

```
def update
  @erratum = Erratum.find(params[:id])

  respond_to do |format|
    if @erratum.update_attributes(params[:erratum])
      flash[:notice] = 'Erratum was successfully updated.'
      format.html { redirect_to(@erratum) }
      format.xml  { head :ok }
      format.js   { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml  { render :xml => @erratum.errors,
                           :status => :unprocessable_entity }
      format.js   { head :unprocessable_entity }
    end
  end
end
```

Discussion

In general when you're toggling attributes with Ajax, you don't care about the response. In those cases this technique works a treat. If the update is dependent on validation, then your best bet is to use a full form and a synchronous request.

Recipe 19

Uploading Images and Creating Thumbnails

Problem

You want to let users upload images (or any file) and generate a variety of thumbnails for use around your site.

Ingredients

- Rick Olson's attachment_fu plugin:

```
$ script/plugin install ↩
    http://svn.techno-weenie.net/projects/plugins/attachment_fu/
```

- Optionally, the AWS:S3 gem:

```
$ gem install aws-s3
```

- One of the following image processing libraries:
  - ImageScience:[16] A light inline-Ruby library that only resizes images. (Wraps the FreeImage library.)
  - RMagick:[17] The grand-daddy, both in terms of advanced image processing features and memory usage. (Wraps the ImageMagick library.)
  - minimagick:[18] It's much easier on memory than RMagick because it runs the ImageMagick command in a shell.

Image processing of this kind is best handled by native code. This means you end up either building a library for your operating system or downloading a pre-built library specific to your operating system. Then you install a Ruby library (gem) that wraps the image processing library with a Ruby API. If you already have one of these installed, go with it.

16. http://seattlerb.rubyforge.org/ImageScience.html
17. http://rmagick.rubyforge.org/
18. http://rubyforge.org/projects/mini-magick/

Solution

Let's say we're building an online jukebox and we need to upload covers for the albums. Thinking about the database first, we could try to cram all the album and cover information into one table. But that can get messy, so we'll split them up into two tables. First, we need an Album model with a few simple attributes:

```
$ script/generate scaffold album title:string artist:string
$ rake db:migrate
```

That gives us everything we need to administer albums. But an album also has one cover:

```
class Album < ActiveRecord::Base
  has_one :cover
end
```

So next we need a database table to store information about the cover: it's size, where it lives, etc. We won't actually store the cover image itself in the database, just its meta-data. Here's the migration for the Cover model:

```
create_table :covers do |t|
  t.integer :album_id, :parent_id, :size, :width, :height
  t.string  :content_type, :filename, :thumbnail
end
```

These database columns are required by the attachment_fu plugin. Again, it's just the information about the cover, not the actual cover image. When a cover image is uploaded, we need to record its location in the covers table and then store the data somewhere. While we're at it, we'd like to generate a few cover image thumbnails to use across our site.

Here's where the attachment_fu plugin really shines. Rather than groveling around at the API level of whatever Ruby image library we have installed, we can declare how we want files to get processed and let attachment_fu work out the details. Here's all we need in our Cover model:

```
class Cover < ActiveRecord::Base

  belongs_to :album

  has_attachment :content_type => :image,
                 :storage => :file_system,
                 :max_size => 500.kilobytes,
                 :resize_to => '384x256>',
                 :processor => "ImageScience",
                 :thumbnails => {
```

```
                :large =>  '96x96>',
                :medium => '64x64>',
                :small =>  '48x48>'
              }

  validates_as_attachment
end
```

In the has_attachment method we tell attachment_fu what to do with the uploaded image. There's a lot packed in here:

- :content_type specifies the content types we allow. In this case, using :image allows all standard image types.
- :storage sets where the actual cover image data is stored. So in truth we could have stored the covers in the database (:db_file), but the file system is easier to manage. Another option is Amazon's S3 service, which we'll look at a bit later.
- :max_size is, not surprisingly, the maximum size allowed. It's always good to set a limit on just how much data you want your app to take in (the default is 1 megabyte).
- :resize_to is either an array of width/height values (for example, :resize_to => [384, 286]) or a geometry string for resizing the image. Geometry strings are more flexible, but not supported by all image processors. In this case, by using the > symbol at the end, we're saying that the image should be resized to 384x286 only if the width or height exceeds those dimensions. Otherwise the image is not resized.
- :processor sets what image processor to use: ImageScience, Rmagick, or MiniMagick. As we haven't specified one, attachment_fu will use whichever library we have installed.
- :thumbnails is a hash of thumbnail names and resizing options. Thumbnails won't be generated if you leave off this option, and you can generate as many thumbnails as you like simply by adding arbitrary names and sizes to the hash.

After configuring how the image gets processes, we call validates_as_attachment to prevent image sizes out of range from being saved. (They're still uploaded into memory, mind you.) As well, because we set an image content type, WinZip files won't be welcome, for example.

OK, now that we have the models created, we turn our attention to the form used to upload the cover image when we create a new Album:

Download FileUploadFu/app/views/albums/new.html.erb

```
<%= error_messages_for :album %>

<% form_for(@album, :html => { :multipart => true }) do |f| %>
  <p>
    <%= label :album, :title %>
    <%= f.text_field :title %>
  </p>
  <p>
    <%= label :album, :artist %>
    <%= f.text_field :artist %>
  </p>
  <p>
    <%= label :album, :cover %>
    <%= f.file_field :uploaded_cover_data %>
    <span class="hint">
      We accept JPEG, GIF, or PNG files up to 500 KB.
    </span>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
```

It's a fairly standard form, but it has two subtle and important bits. First, to allow the form to accept files as POST data, the form_for includes the :multipart => true option. (If you forget to add this, you're in for a long afternoon of debugging.)

Second, the form uses the file_field form helper which generates a "Choose File" button on the form. In this case, the name of the file input field will be album[:uploaded_cover_data]. That means if we were to POST the form, Active Record would expect to set the cover image data into an Album attribute called uploaded_cover_data. There's just one problem: we don't have a column for that in our albums table. And in fact we don't want the image data stored in the albums table. Instead, we'll just create a virtual uploaded_cover_data attribute in our Album model:

```
class Album < ActiveRecord::Base
  has_one :cover
  attr_accessor :uploaded_cover_data
end
```

At this point we can upload an image from the form, but it won't get stored. We haven't done anything with the uploaded cover data. So the

next step is use that cover image data to create a Cover object and associate it with the Album being created. The Album already has the data in the virtual attribute we just created, so we'll let the Album do all the grunt work. This keeps the create action of the controller simple:

Download FileUploadFu/app/controllers/albums_controller.rb

```
def create
  @album = Album.new(params[:album])

  if @album.save_with_cover
    flash[:notice] = 'Album was successfully created.'
    redirect_to(@album)
  else
    render :action => "new"
  end
end
```

Notice we can't just call @album.save here as this would only save the album information. Instead, we've created a save_with_cover method that saves both the album and the cover in a transaction:

Download FileUploadFu/app/models/album.rb

```
def save_with_cover
  cover = Cover.new
  begin
    self.transaction do
      if uploaded_cover_data && uploaded_cover_data.size > 0
        cover.uploaded_data = uploaded_cover_data
        cover.thumbnails.clear
        cover.save!
        self.cover = cover
      end
      save!
    end
  rescue
    if cover.errors.on(:size)
      errors.add_to_base("Uploaded image is too big (500-KB max).")
    end
    if cover.errors.on(:content_type)
      errors.add_to_base("Uploaded image content-type is not valid.")
    end
    false
  end
end
```

OK, so now we're off to the races: we select a cover file using the "Choose File" button on the form, the cover image is uploaded to a file on our server, and the file metadata is stored in the covers database table. We end up with four rows in the covers table: one for the resized original

(parent) image and one for each of the three thumbnails. The thumbnails have their parent_id column set to the primary key of the cover from which they were created.

Each image also has a base filename recorded in the covers table. The public_filename method uses this information to give us the public path to the resized original file, or the thumbnail if passed the name of the thumbnail. Let's inspect our images in the console:

```
$ ruby script/console
>> c = Cover.find :first
=> #<Cover id: 1, album_id: 1, parent_id: nil, size: 72620, width: 201,
     height: 201, content_type: "image/png",
     filename: "foo_fighters.png", thumbnail: nil>
>> c.public_filename
=> "/covers/0000/0001/foo_fighters.png"
>> c.public_filename(:small)
=> "/covers/0000/0001/foo_fighters_small.png"
>> c.public_filename(:medium)
=> "/covers/0000/0001/foo_fighters_medium.png"
>> c.public_filename(:large)
=> "/covers/0000/0001/foo_fighters_large.png"
```

Since we're using the filesystem as storage, our cover image files are stored relative to the $RAILS_ROOT/public directory on our server.[19] The thumbnail files have a suffix that corresponds to the name we used in the :thumbnails hash.

Finally, let's write a view helper so we can easily show covers in various sizes (and linked to the full-size image) around our jukebox site:

Download FileUploadFu/app/helpers/albums_helper.rb

```
module AlbumsHelper

  def cover_for(album, size = :medium)
    if album.cover
      cover_image = album.cover.public_filename(size)
      link_to image_tag(cover_image), album.cover.public_filename
    else
      image_tag("blank-cover-#{size}.png")
    end
  end

end
```

Here's the template for listing all the albums and their covers:

19. The default path prefix for the file system is public/#{table_name}. This can be changed by adding a :path_prefix option to the has_attachment method.

Download FileUploadFu/app/views/albums/index.html.erb

```
<table>
<% for album in @albums -%>
  <tr>
    <td><%= cover_for(album, :large) %></td>
    <td>
      <strong><%= link_to album.title, album %></strong>
      by <%= h album.artist %>
    </td>
  </tr>
<% end -%>
</table>
```

Discussion

One of the big benefits of using attachment_fu is the choice of backend storage systems. Let's say, for example, we want to store all of our covers on Amazon's S3 Web Service.[20] First, we simply change the :storage option on the Cover model to :s3. Then we edit the $RAILS_ROOT/config/amazon_s3.yml configuration file to include our S3 account information:

Download FileUploadFu/config/amazon_s3.yml

```
development:
  bucket_name: your_bucket_name
  access_key_id: your_access_key_id
  secret_access_key: your_secret_access_key

test:
  bucket_name: appname_test
  access_key_id:
  secret_access_key:

production:
  bucket_name: appname
  access_key_id:
  secret_access_key:
```

That's all there is to it! Upload a new cover, and you'll see that the image link on the albums listing points to your image hosted on the S3 server.

Also See

Sorbet 57, *Preserving Files Between Deployments*, on page 249 describes how to keep uploaded images stored on the file system from disappearing between deployments.

20. http://aws.amazon.com/s3

Decouple Your JavaScript with Low Pro

By Adam Keys (http://therealadam.com)
Adam Keys is a connoisseur of code, dachshunds and existentialism jokes.

Problem

Rails gives you some great shortcuts that make building interactive web applications with AJAX really easy. However, the Rails AJAX helpers leave something to be desired when it comes to keeping the JavaScript unobtrusive. How do you structure your JavaScript logic and easily apply it to your pages, and at the same time support users who have JavaScript turned off?

Ingredients

- The Low Pro[21] JavaScript library (lowpro.js) in your public/javascripts directory.

Solution

It turns out that abandoning Rails' AJAX helpers in favor of using JavaScript directly is pretty easy. Doubly so if we use Dan Webb's fantastic Low Pro library. Low Pro lets us write JavaScript *behaviors* that handle the various events an HTML element can emit—events such as onclick, keydown or onsubmit. We can then bind those behaviors to specific HTML elements in our page using CSS selectors. It's a delightful way to incrementally add in the interactive bits of our application.

Let's say we've built a lovely little Rails application for tracking our friends and all their contact info. It doesn't have any JavaScript and thus feels sort of bland, so let's add some fanciness to it.

Before we get started, we need to include the Low Pro JavaScript library. The trick here is that it must be loaded *before* our application-specific JavaScript but *after* the Prototype library. For this reason, we can't use the default JavaScript includes. Instead, we need something like this:

21. http://lowprojs.com

Download Lowpro/app/views/layouts/application.html.erb

```
<%= javascript_include_tag 'prototype', 'effects',
                           'lowpro', 'application' %>
```

The first thing we'll do is take the somewhat boring index page and add some interactivity to it. When we list out our friends, the generated HTML looks like this:

```
<ul class="people_list">
  <li>
    <a href="/people/1">Adam Keys</a>
    <span class="preview">http://therealadam.com</span>
  </li>
</ul>
```

We want to change this so that the person's URL is only shown when we mouse over the person. Now we could add the JavaScript code directly into the HTML, but we'd rather treat the JavaScript as a separate layer on top of our already working application. We're already giving our HTML elements id and class attributes, so we can start attaching behaviors to them.

To do that, first we add a simple Low Pro behavior in our application.js file:

```
var PeoplePreview = Behavior.create({

  initialize: function() {
    this.element.down('.preview').hide();
  },

  onmouseover: function() {
    this.element.down('.preview').show();
  },

  onmouseout: function() {
    this.element.down('.preview').hide();
  }
});
```

Let's take this apart. We're creating a Low Pro behavior by calling Behavior.create. Think of this like a method that creates an object for us, as that is exactly what it does. The methods on the object we pass can have any name we like, but since we're writing a behavior, we should probably throw in some event handlers like onmouseover and onmouseout. If we specify an initialize method, it gets called when the behavior is attached to an actual element.

Within the methods of our behavior, this refers to our behavior object. The Low Pro library arranges for this.element to refer to the element which this behavior was attached to. With the element in hand, we can proceed to do anything we can do in Prototype, such as calling hide() or show() on an element.

Then we attach the behavior to all the HTML elements matching the CSS selector .people_list li like so:

```
Event.addBehavior({
  '.people_list li': PeoplePreview
});
```

The behavior is attached as soon as the DOM is loaded, unobtrusively, so there's no need to call this JavaScript from our views.

Before we proceed, let's clean up what we've already got. Instead of inlining the CSS class to use for previews, we can pass it in to the behavior when it's created. We can then hide the incantation for finding it behind an accessor so our code is nice and DRY. Here's the cleaned up version:

Download Lowpro/public/javascripts/application.js

```
var PeoplePreview = Behavior.create({
  preview: null,

  initialize: function(preview_selector) {
    this.preview = this.element.down(preview_selector);

    this.preview.hide();
  },

  onmouseover: function() {
    this.preview.show();
  },

  onmouseout: function() {
    this.preview.hide();
  }
});
```

It's four lines longer, but half of that is whitespace. Further, changing the code will prove much easier in the future. Note that we can create instance variables if we wish and that arguments to PeoplePreview() are passed into the behavior when it's created.

Now let's make deleting a person more interesting. What we'll do is intercept clicks on the delete button and prompt the user for confirma-

tion. To the user, this will look just like Rails' built-in helper for confirming an action. However, our implementation won't insert JavaScript into the HTML we render, giving us a nice little unobtrusive implementation.

First we'll change our view to render our own form instead of using the button_to helper. Our delete button now looks like this:

Download Lowpro/app/views/people/_person.html.erb

```
<% form_for person,
            :html => {:class => dom_class(person, 'delete'),
                      :method => :delete} do |f| -%>
  <p>
    <%= f.submit "Delete #{person.name}" %>
  </p>
<% end -%>
```

Note that we use the dom_class helper to generate a delete_person class for our delete form. We also have to set the HTML method to :delete to let Rails know we want to emulate a DELETE request. Now that our form is in place, here's the behavior we'll attach to it:

Download Lowpro/public/javascripts/application.js

```
var DeleteConfirmation = Behavior.create({
  onsubmit: function(evt) {
    if (confirm('Really delete this item?')) {
      return true; // Allow the delete
    } else {
      evt.stop();
      return false;
    }
  }
});
```

This behavior is a little different from our first in that we're declaring our handler, onsubmit, as taking an event parameter. (All event handlers are passed this object, we just didn't need it in the first example.) This object contains information on the HTML element that the event occurred on, mouse coordinates at the time of the event, any key-press events and other information. Prototype kindly wraps this all up for us, so we can just treat it as an Event object, rather than the various unfriendly objects various browsers would pass to us.

The crux of our behavior is prompting the user to see if they *really* want to delete this person. If they click OK, our behavior returns true and the browser will continue to submit the form. However, if the user clicks

cancel, we stop the event by calling evt.stop() and returning false from our handler. The form is never submitted and the user's data is safe!

Then we attach this behavior to the entire form. That way we can catch submit events generated by mouse clicks *and* keyboard events (such as the user tabbing through the page and hitting enter on the submit button):

```
Event.addBehavior({
  '.people_list li': PeoplePreview('.preview'),
  '.delete_person': DeleteConfirmation
});
```

Finally, let's add some unobtrusive AJAX. Right now when we edit a person it triggers a full page reload to render the form. Now we'd like to change the Edit link to issue an AJAX request which slips the edit form into the page behind the scenes. That's where Low Pro's Remote behavior comes in. Using Remote, we just need to specify which links and forms are "remote". The behavior takes care of performing the AJAX requests for us.

Let's see how this works out. First, the edit link in question:

Download Lowpro/app/views/people/_person.html.erb

```
<%= link_to "Edit", edit_person_path(person),
            :class => dom_class(person, 'edit') %>
```

We want to make that link a remote AJAX call, if the user has JavaScript turned on. To do so, we just add a behavior rule that matches the link with the edit_person class, like so:

Download Lowpro/public/javascripts/application.js

```
Event.addBehavior.reassignAfterAjax = true;
Event.addBehavior({
  '.people_list li': PeoplePreview('.preview'),
  '.delete_person': DeleteConfirmation,
  '.edit_person': Remote
});
```

The last rule matches our edit link and attaches the Remote behavior to it. Now, when it's clicked, a new Prototype Ajax.Request object is created which requests the URL specified by our link, /people/1/edit for example. We can then use RJS to modify the page, in this case placing an edit form in place of the display markup:

Download Lowpro/app/views/people/edit.rjs

```
page[dom_id(@person)].replace_html :partial => "person_form", :object => @person
page[dom_id(@person, 'edit')].visual_effect :highlight
```

So now our edit form is in place. When the user submits it, we'd like to use AJAX to send the request and then replace the form with the updated person. The great thing about Remote is that it's really a helper on top of Remote.Link and Remote.Form. So you can attach Remote to a link or a form and it will do the right thing.

Since the form we insert into the page has the same class as our link, we don't need another rule to add the behavior. We do however, need to tell Low Pro to reload all its behavior rules after every AJAX request. We do this before we declare our behavior rules by setting Event.addBehavior.reassignAfterAjax to true. For performance reasons, the author of Low Pro doesn't recommend this for all applications. However, for our little application, it's the simplest way to accomplish what we need.

Now our address book is a lot more interesting, and it still works great for folks who don't have JavaScript enabled. Plus, we don't have JavaScript lurking in our HTML, saving us from long nights tracking down logic. Low Pro makes all that easy. Behaviors give you a great way to get everything working without JavaScript first, and then progressively enhance the user experience with AJAX and other JavaScript behaviors.

Also See

We didn't cover things like building composed behaviors or some of the other interesting behaviors that ship with LowPro. If you're itching to learn more, get 'ye to http://lowprojs.com and dig in!

Part V

Design Recipes

Freshening Up Your Models With Scope

By Dan Manges (http://www.dcmanges.com)
Dan Manges is a passionate programmer who focuses on Ruby and Rails development. He enjoys giving back to the community by working on open source projects. After successfully bringing Rails into the enterprise at JPMorgan Chase, he is now a developer with ThoughtWorks.

Problem

You need to use a similar set of database query conditions in multiple scenarios—and even mix and match them with other conditions—all without creating a duplication nightmare.

Ingredients

- The scope_out plugin[22]:

```
$ script/plugin install http://scope-out-rails.googlecode.com/svn/trunk
```

Solution

Let's say we're designing an online newspaper system. Reporters draft articles and can schedule them to be publicly viewable later. So we need to make sure only the finished and ready-for-publish articles are shown on the site. Clearly this leads to a maintenance headache if we try to filter articles in the controller:

```
class ArticlesController < ApplicationController

  def index
    @articles = Article.find(:all,
                  :conditions => ["draft = ? AND publish_date <= ?",
                                  false, Time.now])
  end

  def show
    @article = Article.find(params[:id],
                 :conditions => ["draft = ? AND publish_date <= ?",
                                 false, Time.now])
```

22. http://code.google.com/p/scope-out-rails/

```
  end
end
```

So the first step toward removing the duplication is to push the conditions back into the Article model where they belong. After all, the Article model is the sole authority on what it means for an article to be publicly viewable. This is called *encapsulation* and it's our best defense against brittle (and ugly) code. Here's what our refactored Article model looks like:

```
class Article < ActiveRecord::Base

  def self.find_all_publicly_viewable
    find(:all, :conditions => ["draft = ? AND publish_date <= ?",
                               false, Time.now])
  end

  def self.find_publicly_viewable(id)
    find(id, :conditions => ["draft = ? AND publish_date <= ?",
                             false, Time.now])
  end
end
```

All we did was move code around—the details are now hidden behind custom finder methods. It doesn't seem like much progress, but it cleans up our controller considerably:

```
class ArticlesController < ApplicationController

  def index
    @articles = Article.find_all_publicly_viewable
  end

  def show
    @article = Article.find_publicly_viewable(params[:id])
  end
end
```

That's a good start. However, we can do better, and we will! Our Article model still has duplication in the :conditions option. Also, our custom finder methods are currently limited in that we can't pass in additional find options for ordering, limiting, and so on.

The next refactoring step is to use with_scope to surround a single find method:

```
class Article < ActiveRecord::Base

  def self.find_publicly_viewable(*args)
    with_scope(:find =>
```

```
                {:conditions => ["draft = ? AND publish_date <= ?",
                                 false, Time.now]}) do
      find(*args)
    end
  end
end
```

The with_scope method simply uses the options passed to it to set the scope of the database operations within its block. In other words, it *scopes* the find operation to all publicly viewable articles. That also means we can pass additional options into our custom finder without bothering to merge our options with the default options. For example, now our controller can use the same custom finder, but with different options depending on the action:

```
class ArticlesController < ApplicationController

  def index
    @articles = Article.find_publicly_viewable(:all,
                          :order => 'publish_date DESC')
  end

  def show
    @article = Article.find_publicly_viewable(params[:id])
  end
end
```

Now let's suppose we need to get a count of the total number of publicly viewable articles or calculate the average number of pages (again, just for publicly viewable articles). The find method in the block of with_scope won't work. Instead, we need to extract the with_scope part of the finder method into its own method so we can reuse it. Here's the revised model:

```
class Article < ActiveRecord::Base

  def self.find_publicly_viewable(*args)
    with_publicly_viewable { find(*args) }
  end

  def self.calculate_publicly_viewable(*args)
    with_publicly_viewable { calculate(*args) }
  end

  def self.with_publicly_viewable
    with_scope(:find =>
                {:conditions => ["draft = ? AND publish_date <= ?",
                                 false, Time.now]}) do
      yield
```

```
    end
  end
end
```

The with_publicly_viewable method just sets the scope, then yields control over to a block. Inside the block we can run a find or a calculate method and know that the resulting database operations are properly scoped. So to get a count or average of all publicly viewable articles, we'd use:

```
Article.calculate_publicly_viewable(:count, :all)
Article.calculate_publicly_viewable(:avg, :pages)
```

We can also stack with_scope blocks to mix and match conditions. For example, let's say we have a scope for articles that are premium (you have to log in to read them) in a method called with_premium:

```
def self.with_premium
  with_scope(:find => {:conditions => {:premium => true}}) do
    yield
  end
end
```

Then we could combine the scope to define publicly viewable premium articles like so:

```
def self.find_publicly_viewable_premium_articles(*args)
  with_publicly_viewable do
    with_premium do
      find(*args)
    end
  end
end
```

ActiveRecord will continue merging conditions throughout the chain of with_scope blocks. In other words, if we're applying more than one with_scope block and specifying an option in the find call that was also in a with_scope block, the value in the find wins. All options can be overwritten except for the :conditions option, which merges using an AND operator.

This is a major improvement over the original code! Indeed, it's about as DRY as we can make our model using standard Rails facilities. However, we can further simplify the definition of these scopes using the scope_out plugin. Here's our revised model using scope_out:

```
class Article < ActiveRecord::Base

  scope_out :publicly_viewable do
    { :conditions => ["draft = ? AND publish_date <= ?", false, Time.now] }
```

```
  end

  scope_out :premium
  # same as: scope_out :premium, :conditions => {:premium => true}

  combined_scope :publicly_viewable_premium,
                 [:publicly_viewable, :premium]
end
```

The publicly_viewable scope needs to be defined in a block due to the dynamic time. If defined without the block, the time would always be set to the time when the Article class was loaded.

After defining a scope, the plugin automatically creates the with_*, find_*, and calculate_* methods. This gives us all the scoping we had before, plus a few handy methods. Here are some examples:

```
>> Article.find_publicly_viewable(:all)
=> [#<Article id: 3,...]
>> Article.find_publicly_viewable_by_title('The Title')
=> #<Article id: 3,...
>> Article.find_publicly_viewable_premium(:all)
=> [#<Article id: 3,...]
>> Article.calculate_publicly_viewable_premium(:count, :all)
=> 6
```

Fresh, clean and (most important) easy to maintain.

Discussion

When defining class-level finder methods, they also work through association proxies. For example, let's say we have a Reporter model that has_many :articles. With the scopes defined in this recipe, we could do:

```
Reporter.find_by_name('Dan').articles.find_publicly_viewable(:all)
```

That functionality works with stock Rails; it is not added by the scope_out plugin.

It may be tempting to use with_scope in an around_filter in your controller, or override the default find to apply this scope. However, these techniques are not recommended as they hide your scope and add unexpected behavior to a method familiar to all Rails developers.

Also See

- Thanks to Chris Wanstrath for the blog post "with_scope with scope". [23]

23. http://errtheblog.com/post/41

Recipe 22

Keeping Forms Dry and Flexible

By Mike Mangino (http://www.elevatedrails.com)
Mike Mangino is the founder of Elevated Rails (http://www.elevatedrails.com). He lives in Chicago with his wife Jen and their two Samoyeds.

Problem

Your non-view code is DRY and beautiful, but you cringe every time you look at your forms. You have variations of the same few lines all over the place. You want to move your forms to a standards-based layout—and perhaps even change the layout in one place later—but you can't stand the thought of changing all that code.

Solution

It's quick and painless to DRY up our forms using a custom form builder, and get lots of other goodies along the way. It turns out we've been using a form builder all along, without even knowing it. Here's one:

Download DryUpYourForms/app/views/people/new.html.erb

```
<% form_for(@person) do |f| -%>
  <p>
    <%= label :person, :first_name %>
    <%= f.text_field :first_name %>
  </p>
  <p>
    <%= label :person, :last_name %>
    <%= f.text_field :last_name %>
  </p>
  <p>
    <%= label :person, :bio %>
    <%= f.text_area :bio %>
  </p>
  <p>
    <%= f.submit 'Create' %>
  </p>
<% end -%>
```

The f block parameter that form_for yields is a FormBuilder instance. The default builder doesn't do very much, but it does let us skip using the

@person object in every form field. Some duplication has been removed, but there's still quite a bit left.

Let's start drying this up by creating a custom FormBuilder to evaporate all those label and paragraph tags. Here's a simple implementation:

Download DryUpYourForms/lib/label_form_builder.rb

```
class LabelFormBuilder < ActionView::Helpers::FormBuilder

  helpers = field_helpers +
            %w(date_select datetime_select time_select) -
            %w(hidden_field label fields_for)

  helpers.each do |name|
    define_method(name) do |field, *args|
      options = args.last.is_a?(Hash) ? args.pop : {}

      @template.content_tag(:p, label(field) +
                                super(field, options))
    end
  end
end
```

This isn't much code, but it does a lot for us. First and foremost, form builders are subclasses of ActionView::Helpers::FormBuilder. This means that our form builder already knows how to create the standard form elements (input fields, for example). We just want to wrap each standard element with a paragraph and add a label.

The code starts out by building up the helpers variable with the names of form helpers we want to decorate: the default form helpers, plus a few that aren't included in the defaults, and minus a few that don't need labels. Then we loop through the helper names and use define_method to create a method for each one. If we wanted to define one of these methods explicitly, it would look like this:

```
def text_field(field, *args)
  @template.content_tag(:p, label(field) + super)
end
```

The @template variable inside our form builder is a reference to the view context in which a form element is being executed. Calling the content_tag method on the template just slaps in the content (our label and input field) surrounded by a tag (the paragraph).

The label method used here is slightly different than the one we used in our original form. Specifically, this version of label doesn't need the object because the form builder already has a reference to the object

instance that was passed to the form_for method (the @person). Simply by giving it the field, it'll generate the label tag with an appropriate for attribute (person_first_name, for example). So to summarize, we're wrapping each form element in a p tag *and* creating an appropriately-named label tag for it.

Now we can reduce our original form to this:

Download DryUpYourForms/app/views/people/new.html.erb

```
<% form_for(@person, :builder => LabelFormBuilder) do |f| -%>
  <%= f.text_field :first_name %>
  <%= f.text_field :last_name %>
  <%= f.text_area  :bio %>
  <%= f.submit 'Create' %>
<% end -%>
```

That's definitely an improvement, but we lost some flexibility. It would be nice if we could override the field labels, for instance. While we're at it, we might as well add better highlighting of fields and error messages. To do that, let's leave our LabelFormBuilder and create a new ErrorHandlingFormBuilder:

Download DryUpYourForms/lib/error_handling_form_builder.rb

```
class ErrorHandlingFormBuilder < ActionView::Helpers::FormBuilder

  helpers = field_helpers +
    %w(date_select datetime_select time_select collection_select) -
    %w(label fields_for)

  helpers.each do |name|
    define_method name do |field, *args|
      options = args.detect {|argument| argument.is_a?(Hash)} || {}
      build_shell(field, options) do
        super
      end
    end
  end

  def build_shell(field, options)
    @template.capture do
      locals = {
        :element => yield,
        :label   => label(field, options[:label])
      }
      if has_errors_on?(field)
        locals.merge!(:error => error_message(field, options))
        @template.render :partial => 'forms/field_with_errors',
                         :locals  => locals
      else
```

```
        @template.render :partial => 'forms/field',
                         :locals  => locals
      end
    end
  end

  def error_message(field, options)
    if has_errors_on?(field)
      errors = object.errors.on(field)
      errors.is_a?(Array) ? errors.to_sentence : errors
    else
      ''
    end
  end

  def has_errors_on?(field)
    !(object.nil? || object.errors.on(field).blank?)
  end
end
```

Here we see our friend @template again. Using render, we render one of two templates depending on whether the field has errors. Due to the way we're using @template, instance variables won't be passed to the templates. So we have to use the locals hash to pass in the options. The result of rendering the template is captured in the template.

This version is quite an improvement now that we've moved our presentation into templates where it belongs. Now our form handling logic is separate from the layout and style of the views. Inside our form field template we then access the local variables:

Download DryUpYourForms/app/views/forms/_field.html.erb

```
<p>
  <span class="field_label">
    <%= label %>
  </span>
  <span class="form_field">
    <%= element %>
  </span>
</p>
```

The template for a form element with errors is similar, but includes the error below each form element and a CSS class for painting it a jarring color:

Download DryUpYourForms/app/views/forms/_field_with_errors.html.erb

```
<p>
  <span class="field_label">
    <%= label %>
```

```
  </span>
  <span class="form_field">
    <%= element %>
    <span class="form_error_message">
      <%= error %>
    </span>
  </span>
</p>
```

Now that our forms are nice and DRY, let's clean up having to constantly specify a :builder parameter when calling form_for. To do that, we just need a helper method that automatically adds the builder option for us:

Download DryUpYourForms/app/helpers/application_helper.rb

```
def error_handling_form_for(record_or_name_or_array, *args, &proc)
  options = args.detect { |argument| argument.is_a?(Hash) }
  if options.nil?
    options = {:builder => ErrorHandlingFormBuilder}
    args << options
  end
  options[:builder] = ErrorHandlingFormBuilder unless options.nil?
  form_for(record_or_name_or_array, *args, &proc)
end
```

Whew, it took a while to get here, but the reward is a bone-dry form:

Download DryUpYourForms/app/views/people/new.html.erb

```
<% error_handling_form_for(@person) do |f| -%>
  <%= f.text_field :first_name %>
  <%= f.text_field :last_name, :label => 'Family Name' %>
  <%= f.text_area  :bio %>
  <%= f.submit 'Create' %>
<% end -%>
```

The internals of a form builder can feel really messy. It's meta-programming, subclassing, and groveling in view internals all rolled into one. Thankfully, it's localized to just one file. The big payoff comes when you want to change the way all your forms look (and handle errors). You just tweak the form builder and away you go.

Discussion

You could easily subclass this form builder if you wanted to have different looks for different forms. You could also dynamically change forms based upon an input option.

Prevent Train Wrecks with Delegate

By Hugh Bien (http://hughbien.com)
Hugh is a web programmer who likes working with agile languages. He uses Rails at his day job and keeps himself busy with fun side projects.

Problem

ActiveRecord associations make it easy to traverse model relationships: just add one more dot. But go too far, and you often end up with method chains that access attributes through a relationship, like this:

```
account.subscription.free?
account.subscription.last_payment.overdue?
```

Some people call this object-oriented programming. We call it a train wreck. If the details of how a subscription handles its last payment change, for example, the whole thing goes off the rails. So how do you clean this up?

Solution

One solution is to encapsulate far-reaching attributes in methods that delegate to other models, like so:

```
class Account < ActiveRecord::Base
  has_one :subscription

  def free?
    self.subscription.free?
  end
end
```

But there's an easier way. We can use the delegate method to keep one object from knowing too much about the objects it's related to. (It's a *shy* object.) Instead of defining a free? method in our Account class, we can just delegate it straight to the account's subscription.

Download PreventTrainWrecksWithDelegate/app/models/account.rb

```
class Account < ActiveRecord::Base
  has_one :subscription
  delegate :free?, :to => :subscription
end
```

Then given an Account object, we can just call the free? method directly:

```
account.free?
```

The delegate method is very easy to use, but there's more to it under the hood. We can delegate multiple methods to another object, for example:

Download PreventTrainWrecksWithDelegate/app/models/account.rb

```
class Account < ActiveRecord::Base
  has_one :subscription
  delegate :free?, :paying?, :to => :subscription
end
```

And we can also use delegate to traverse through more than one association:

Download PreventTrainWrecksWithDelegate/app/models/account.rb

```
delegate :overdue?, :to => "subscription.last_payment"
```

Finally, sometimes an account might exist that does not have a subscription. In this case, we'll get a NoMethodError of a Nil object (sometimes called the whiny nil). We can prevent this with a little hack:

Download PreventTrainWrecksWithDelegate/app/models/account.rb

```
delegate :free?, :paying?, :to => "subscription.nil? ? false : subscription"
```

Let's see how that works in the console without a subscription:

```
$ ruby script/console
>> a = Account.new
=> #<Account id: nil ...>
>> a.subscription
=> nil
>> a.free?
=> false
```

If no subscription exists, then free? returns false rather than nil.

Now if we assign a subscription to the account, it delegates the call as expected:

```
>> a.subscription = Subscription.new(:free => true)
=> #<Subscription id: nil ...>
>> a.free?
=> true
```

Using this simple technique, we can reduce unnecessary coupling across our codebase. Instead of having account.subscription.last_payment.overdue? gumming up the works, we can use the shy account.overdue? method instead. That way, any changes to how our associations work will require just one change in our call to delegate.

Discussion

If you find yourself using delegate on a frequent basis, it may be a *smell*, as they say.

Creating Meaningful Relationships Through Proxies

By Mike Mangino (http://www.elevatedrails.com)
Thanks to Mike Mangino for the reloading tip in this recipe.

Problem

You find yourself writing lots of custom finder methods to constrain your model associations based on a variety of scenarios—finding all paid registrations for a given event, for example. As a result, your model code has gradually turned into something you're not proud of and you'd like to clean it up.

Solution

Let's walk through a series of code refactorings, starting with the following relationship:

```
class Event < ActiveRecord::Base
  has_many :registrations
end
```

We'd like to add some code that returns just the registrations that have been paid. Where do we put that code? To answer that, let's first dig deeper into the association:

```
$ ruby script/console
>> e = Event.find :first
=> #<Event id: 1...>
>> e.registrations
=> []
>> e.registrations.class
=> Array
```

Here's where things get interesting. The registrations method returns something that looks like an array, and acts like an array, but in fact it's *not* an array. It's really an *association proxy* in disguise. Among its other special powers, it lets us call Active Record class methods on the association:

```
>> e.registrations.create(:name => 'Fred', :paid_at => Time.now, :price => 5.00)
=> #<Registration id: 1...>
```

```
>> e.registrations.find(:all, :conditions => "paid_at is not null")
=> #<Registration id: 1...>

>> e.registrations.sum(:price)
=> #<BigDecimal:190c92c,'0.5E1',4(8)>
```

In other words, the association is proxying method calls—create, find, sum, etc.—through to the Registration class. But, and here's the really important part, it scopes any database operations to the root object (an event in this case). Here's the SQL generated by the methods we just ran:

```
INSERT INTO `registrations` (`event_id`, `name`, `price`, `paid_at`)
  VALUES(1, 'Fred', '5.0', '2007-12-14 09:42:15')
SELECT * FROM `registrations` WHERE (registrations.event_id = 1
  AND (paid_at is not null))
SELECT sum(price) AS sum_price FROM `registrations`
  WHERE (registrations.event_id = 1)
```

So if we can proxy calls through to the Registration class, then we can start by putting the conditions for finding paid registrations in a class method:

```
class Registration < ActiveRecord::Base
  belongs_to :event

  def self.paid
    find :all, :conditions => "paid_at is not null"
  end
end
```

And that lets us find all paid registrations for a specific event:

```
>> e.registrations.paid
=> [#<Registration id: 1...]
```

However, since we're always scoping paid registrations to the event, we can actually transform the code into a well-named association (paid_registrations) on the Event class itself:

```
class Event < ActiveRecord::Base

  has_many :registrations

  has_many :paid_registrations,
           :class_name => "Registration",
           :conditions => "paid_at is not null"
end
```

It's the same underlying database operation, but the call is a bit more meaningful:

```
>> e.paid_registrations
=> [#<Registration id: 1...]
```

Now we have the code where we want it, but it's inconvenient to maintain two associations. We can actually do one better. It turns out we can pass a block to an association which defines methods just for that association. Here's the same code, this time using just one association:

```
class Event < ActiveRecord::Base
  has_many :registrations do
    def paid
      find(:all, :conditions => "paid_at is not null")
    end
  end
end
```

Now the call looks like something we've seen before:

```
>> e.registrations.paid
=> [#<Registration id: 1
```

With all design decisions, there's a trade-off. Just calling e.registrations is cached—it'll only hit the database the first time. However, calling e.registrations.paid isn't cached. It falls out of bounds of Active Record default caching. But we can fix that easy enough:

```
class Event < ActiveRecord::Base
  has_many :registrations do
    def paid(reload=false)
      @paid_registrations = nil if reload
      @paid_registrations ||= find(:all, :conditions => "paid_at is not null")
    end
  end
end
```

Now we have a simple, but effective, association proxy cache that we can expire by passing in true. Named associations like this go a long way toward making code more expressive and maintainable.

Part VI

Asynchronous Recipes

Processing an Asynchronous Workflow

By Jonathan Dahl (http://slantwisedesign.com)
Jonathan Dahl is a Founding Partner at Slantwise Design, a web application development shop in Minnesota. Besides 20-odd Rails projects, Jonathan has done extensive work with video transcoding, and just released Zencoder, a distributed video processing system built with Ruby (http://zencoder.tv).

Problem

Your application needs to execute a time-consuming process such as video transcoding or large PDF generation. The process will take more than a few seconds, so you can't do it synchronously within the lifecycle of an HTTP request. But you want the processing to begin immediately, so you can't just trigger it with a nightly or hourly cron job. And here's the kicker: You also want to track the status of the job as it transitions from one state to the next.

Ingredients

- Scott Barron's acts_as_state_machine plugin:

```
$ script/plugin install ↩
    http://elitists.textdriven.com/svn/plugins/acts_as_state_machine/trunk
```

- The simple-daemon gem:

```
$ gem install simple-daemon
```

Solution

Let's take the example of PDF generation. Requests for PDF updates come in and we need to get them stamped as soon as possible. On a particularly busy day when PDFs get queued up, we also need to track their status. It seems like a fairly difficult task, but with a couple off-the-shelf ingredients we'll be up and running in no time. We'll use a simple Ruby daemon (simple-daemon) to poll our database looking for new PDF generation jobs and the acts_as_state_machine plugin to manage the workflow states.

First we need a model that represents the work to be done, so let's start by creating a Pdf model and migration:

```
$ script/generate model pdf title:string state:string ↩
    version:integer priority:integer processing_error_message:string
$ rake db:migrate
```

Of particular note are the state and version columns:

- The state column will be used by acts_as_state_machine to track what's currently happening with a PDF.
- The version column is a special Active Record column used for optimistic locking. It ensures that if two processes access the same row and try to save competing edits, the second edit will fail with an ActiveRecord::StaleObjectError exception. We'll handle that shortly.

Next, in our Pdf model we set up acts_as_state_machine to transition between states when certain events are fired (see Recipe 17, *Creating a Wizard*, on page 83 for a state machine refresher):

Download AsyncWorkflow/app/models/pdf.rb

```
class Pdf < ActiveRecord::Base

  acts_as_state_machine :initial => :pending

  state :pending
  state :processing
  state :complete
  state :error

  event :start_pdf_generation do
    transitions :from => :pending, :to => :processing
  end

  event :finish_pdf_generation do
    transitions :from => :processing, :to => :complete
  end

  event :processing_error do
    transitions :from => :processing, :to => :error
  end

  def self.find_for_pdf_generation
    find(:first,
         :conditions => "state = 'pending'",
         :order => "priority, created_at")
  end

  def generate_pdf
    logger.info("Generating #{title} PDF...")
    # Insert your long-running code here to generate PDFs.
```

```
  end

end
```

A Pdf starts in the pending state. When a start_pdf_generation event occurs, for example, the Pdf transitions to the processing state. In this way it goes from pending through to complete provided there are no errors. We've also written a custom finder to fetch all the pending PDFs, ordered by priority.

Now we need a side dish of code that will drive our little PDF state machine. As it's not a model, we'll just place it in the lib directory:

Download AsyncWorkflow/lib/generate_pdf.rb

```
require File.dirname(__FILE__) + '/../app/models/pdf.rb'

class GeneratePdf

  def self.run
    begin
      pdf = Pdf.find_for_pdf_generation
      raise ActiveRecord::RecordNotFound if pdf.nil?

      pdf.start_pdf_generation!
      pdf.generate_pdf
      pdf.finish_pdf_generation!

    rescue ActiveRecord::StaleObjectError
      # do nothing
    rescue ActiveRecord::RecordNotFound
      sleep 10
    rescue
      return unless pdf
      pdf.processing_error!
      pdf.update_attributes(:processing_error_message => "unknown error: #{$!}")
    end
  end

end
```

The single method in this class, run, first tries to pick a PDF off the pile using the method we wrote to return the next highest-priority PDF. If a PDF is pending, the run method marks the PDF as being in-process by triggering the start_pdf_generation event. Then it carries on with the real work of generating the PDF. Finally the finish_pdf_generation event is triggered, and our PDF updates to the complete state.

That's the happy path. If there's no pending PDF, the run method sleeps for 10 seconds and then checks again. If a StaleObjectError is raised—

where two processes tried to save competing changes to the same PDF record—the second process will fail silently and move on to the next PDF. Finally, if an unknown exception is encountered, the PDF is moved to the error state and the exception message is saved in the processing_error_message column.

The last piece of the puzzle is the daemon process that calls our run method in a loop. It's a script we'll run from the command line, so it goes in the scripts/pdf_generator file:

Download AsyncWorkflow/script/pdf_generator

```
#!/usr/bin/env ruby

RAILS_ENV = ARGV[1] || 'development'

require File.dirname(__FILE__) + '/../config/environment.rb'

class PdfGeneratorDaemon < SimpleDaemon::Base

  SimpleDaemon::WORKING_DIRECTORY = "#{RAILS_ROOT}/log"

  def self.start
    loop do
      GeneratePdf.run
    end
  end

  def self.stop
    puts "Stopping PDF Generator..."
  end
end

PdfGeneratorDaemon.daemonize
```

This script uses the SimpleDaemon::Base class to manage a PID file. When the daemon is started, the log/pdf_generator.pid file is created and the process ID of the daemon process is slipped inside. When the daemon is stopped, the PID is read from the log/pdf_generator.pid and the corresponding process is stopped.

Before we run all this, we need to add two requirements which we'll throw in a Rails initializer file:

Download AsyncWorkflow/config/initializers/generate_pdf.rb

```
require 'simple-daemon'
require 'generate_pdf'
```

OK, now let's fire up the daemon in development mode:

```
$ ruby script/pdf_generator start development
Daemon started.
```

Then if we throw a PDF into the database using the console, the PDF gets picked up and processed behind the scenes for us:

```
$ ruby script/console
>> p = Pdf.create(:title => "Advanced Rails Recipes", :priority => 1)
>> p.state
=> "pending"
>> p.reload
>> p.state
=> "complete"
```

When we're all done, it's polite to give the daemon a rest:

```
$ ruby script/pdf_generator stop development
```

Finally, to put all this into production, we'll just add the following custom tasks to our Capistrano recipe in the config/deploy.rb file, and run them automatically after the standard deployment chores are done:

```
task :start_pdf_generator :roles => :app do
  run "ruby #{current_path}/script/pdf_generator start production"
end

task :stop_pdf_generator :roles => :app do
  run "ruby #{current_path}/script/pdf_generator stop production"
end

after(:deploy) { start_pdf_generator; stop_pdf_generator }
```

Then sit back, relax, and let the hamsters do all the work...

Discussion

Workflows often include multiple states (or stages) with various events leading to each state. The rules can get fairly complex quickly, and acts_as_state_machine really shines in these scenarios. In this case, we used it as an effective way to model a simple workflow with a daemon processing it asynchronously. If you have multiple servers running your main application, you can run a processing daemon on each server. The downside, if you'd call this a downside, is that your processor is tightly bundled to your Rails application. It integrates at the model level, using the Pdf model in this case to find and process jobs.

Another approach to asynchronous processing—and there are many—is to integrate with a message queue. Amazon's Simple Queuing Service

(SQS) is a good candidate. Whenever a new PDF is available for processing, your Rails application will send a message to SQS that identifies the PDF ID, the location of the file, and instructions for processing. Your processing scripts poll your SQS queue for new jobs; when they find one, they do their processing and then respond to your Rails application synchronously through REST.

Finally, there are at least three additional improvements you should consider:

- Check for lost jobs. If a job enters the processing state but doesn't finish, it will be lost in limbo. Add a processed_at column to the Pdf model and set the state machine to populate processed_at with the current time when the document enters the processing state. You'll also need to change the find_for_pdf_generation method to look for jobs with a state of processing that were marked as processed_at more than *N* minutes or hours ago.
- Receive email notifications of unhandled exceptions. You probably use the exception_notifier plugin for you main Rails app; patch into that plugin, or write your own exception code.

Also See

For an example of how to monitor the daemon process, see Recipe 67, *Monitoring (and Repairing) Processes with Monit*, on page 285.

Off-Loading Long-Running Tasks to BackgrounDRb

By Gregg Pollack and Hemant Kumar (http://www.railsenvy.com,http://gnufied.org)
Gregg Pollack lives in Orlando, Florida where he runs the Orlando Ruby Users Group, writes on his blog Rails Envy, and is always hunting for more Rails projects for his company Patched Software. Hemant Kumar is the current maintainer of the BackgrounDRb library.

Problem

You have an action that starts a long-running task—say, around 30 seconds. The task is something that the user wants to wait around for, but you don't want to tie up the entire Rails process for one request/response cycle and risk the user inching his mouse towards the dreaded refresh button. Instead, you'd like to off-load the task to a background process, respond to the original web request immediately, and continually give the user updates on the status of the task.

Ingredients

- The BackgrounDRb plugin:[24]

```
$ script/plugin install http://svn.devjavu.com/backgroundrb/trunk/
```

Solution

Let's say our application is a virtual mall for boutique shops. Customers buy stuff throughout the day, and shop owners needs to charge their customers in batch to optimize the payment process. On a good sales day it might take a few seconds to run all the charges through the system, and the conventional web request/response cycle gets really clunky. And every time a shop owner presses the button to run the charges, we have one less Rails process available for other work. So we need a way to run the billing code in a separate process, and then periodically update the status in the browser.

All this talk of background tasks sounds like a lot of work, but it's surprisingly straightforward with BackgrounDRb. The BackgrounDRb

24. http://backgroundrb.rubyforge.org/

server is just a process that has access to our Rails environment. We queue up work through a MiddleMan object living in our Rails application, and the MiddleMan delegates the work to workers, leaving our Rails process free to take new web requests. And just like any good middle man, we can ask it for an update on worker status.

After installing the BackgrounDRb plugin, we need to configure it:

```
$ rake backgroundrb:setup
```

This gives us three files and a directory:

- The config/backgroundrb.yml file contains the basic configuration required to run the BackgrounDRb server. The contents of the file looks like this:

  ```
  ---
  :backgroundrb:
    :port: 11006
    :ip: 0.0.0.0
  ```

  The BackgrounDRb server will listen on the specified port and ip. By default all workers will be loaded in the development environment which can be changed using:

  ```
  ---
  :backgroundrb:
    :environment: production
  ```

- The script/backgroundrb script starts, stops, and restarts the BackgrounDRb server process.
- The test/bdrb_test_helper.rb is a helper file for testing our workers.
- The lib/workers directory houses the worker code.

Lets go ahead and create a worker to handle our long-running billing chores:

```
$ script/generate worker billing
```

This generates a skeleton lib/workers/billing_worker.rb file like this:

```
class BillingWorker < BackgrounDRb::MetaWorker
  set_worker_name :billing_worker

  def create(args = nil)
  end
end
```

Obviously we need to add code inside the create method. To get the hang of things, lets start with a some logging and a delay:

```
class BillingWorker < BackgrounDRb::MetaWorker
  set_worker_name :billing_worker

  def create(args=nil)
    args.each do |customer_id|
      logger.info "Billing customer #{customer_id}..."
      sleep(3)
    end
    logger.info "All Done"
    exit
  end
end
```

Now that we've created a worker, we can start the BackgrounDRb server:

```
$ script/backgroundrb start
```

Then we'll jump straight into the console and start our worker:

```
$ ruby script/console
>> key = MiddleMan.new_worker(:worker => :billing_worker,
  :job_key => "abc123", :data => [1, 2, 3])
=> "abc123"
```

This creates a BillingWorker in a new process and invokes the create method, passing in our array of customer ids (the array we used for the :data option). As we have three customers with a 3-second delay between each, this method should run for 9 seconds. But we don't have to wait around—the new_worker method returns as soon as the worker has started and hands us back the job key for future reference.

Now if we peek in the log/backgroundrb.log file we'll see:

```
Billing customer 1...
Billing customer 2...
Billing customer 3...
All Done
```

This is all hunky-dory, but it's currently a one-way street through the MiddleMan and, well, the work isn't very interesting. So our the next step is to actually bill each customer and keep track of the status along the way. Here's our revised worker:[25]

Download BackgrounDRb/lib/workers/billing_worker.rb

```
class BillingWorker < BackgrounDRb::MetaWorker
  set_worker_name :billing_worker
  set_no_auto_load(true)
```

25. If you change the source of one of your workers, you will need to stop and start your BackgrounDRb server to see the changes.

```
  def create(args=nil)
    register_status(:percent_complete => 0)
    args.each_with_index do |customer_id, index|
      c = Customer.find(customer_id)
      c.bill!
      percent_complete = ((index + 1) * 100) / args.length
      logger.info "Billing is #{percent_complete}% complete..."
      register_status(:percent_complete => percent_complete)
    end
    exit
  end

end
```

We have access to our Customer model from inside the worker, which means we can neatly tuck the billing logic behind the bill! method. As each customer is billed, we use the register_status method to accumulate how far along the worker is in the :percent_complete status variable.

We've also slipped in a call to the set_no_auto_load method. By default each of the workers found in the lib/workers directory will be started in a separate process when the BackgrounDRb server starts. We don't need to do this automatically, so we've disabled auto-loading.

That takes care of the worker. Next we need to plug all this into our Rails application.

Lets assume we have a form that lists all the customers who haven't been billed yet and includes a checkbox next to each one indicating whether we want to bill them. When we post the form, we want to call the MiddleMan.new_worker method and toss all the customer ids that need billing to our worker. Rather than clutter the controller action with BackgrounDRb details, we'll encapsulate the starting and statusing in methods of our Shop model. Here's the full model:

Download BackgrounDrb/app/models/shop.rb

```
class Shop < ActiveRecord::Base
  has_many :customers
  validates_presence_of :name, :login, :password

  def unpaid_customers
    customers.find(:all, :conditions => "last_billed_at is null")
  end

  def start_billing(customers_to_charge)
    MiddleMan.new_worker(:worker => :billing_worker,
                         :job_key => self.id,
```

```
                          :data => customers_to_charge)
  end

  def self.billing_status(job_key)
    status = MiddleMan.ask_status(:worker => :billing_worker,
                                  :job_key => job_key)
    status[:percent_complete]
  end

end
```

Then over in the create action of our ChargesController we just roll up all the customer IDs to charge and start the billing:

Download BackgrounDrb/app/controllers/charges_controller.rb

```
def create
  customers_to_charge = []
  params[:charge_customer].each do |customer_id, charge|
    customers_to_charge << customer_id if charge == "yes"
  end

  session[:bill_job_key] = @current_shop.start_billing(customers_to_charge)

  redirect_to :action => 'check_bill_status'
end
```

Just like in the console, we get the job key back as soon as the worker has started. In this case, we stash it away in the session. Then we immediately redirect to the check_bill_status action and use the job key to ask our worker for a status:

Download BackgrounDrb/app/controllers/charges_controller.rb

```
def check_bill_status
  @percent_complete = Shop.billing_status(session[:bill_job_key])

  if request.xhr?
    if @percent_complete == 100
      render :update do |page|
        flash[:notice] = "Billing is complete!"
        session[:bill_job_key] = nil
        page.redirect_to :action => "index"
      end
    else
      render :update do |page|
        page[:billingStatus].setStyle :width => "#{@percent_complete * 2}px"
        page[:billingStatus].replace_html "#{@percent_complete}%"
      end
    end
  end
end
```

At this point, the action falls right through to a template that shows no progress, but starts a periodic remote call back to the check_bill_status method to poll for a status every three seconds:

Download BackgrounDrb/app/views/charges/check_bill_status.html.erb

```
<h1>Billing in Progress</h1>

<div id="billingStatus" class="progress">
</div>

<%= periodically_call_remote :url => "check_bill_status", :frequency => 3 %>
```

Each time the action is called, it asks the Shop for its billing status and uses an RJS update block to update the billingStatus DIV progress bar. When the billing is complete, we clear out the job key and use RJS to issue a full redirect back to the list of pending charges.

The result is a fairly transparent solution for firing up separate processes to handle long-running tasks and continuously reporting their status back to the user.

Now let's say we also want to use BackgrounDRb to perform a particular off-line task, such as sending receipt e-mails to our customers. Unlike the billing task, we basically just want to fire off a request to a specific worker method without waiting for the result. To do that, we can use the ask_work method:

```
MiddleMan.ask_work(:worker => :billing_worker,
                   :worker_method => :mail_receipts,
                   :data => customer_ids)
```

Except on a really busy day we might have multiple shop owners all sending emails at the same time. And if the worker is already busy when a new task arrives, the work won't get queued and pretty soon customers start phoning up asking what happened to their receipt. Instead we want our mail_receipts tasks to pile up and get worked off one after the other, reliably. The solution is simple: We just configure the built-in thread pool with a worker size of 1 and use the thread_pool.defer method:

```
class BillingWorker < BackgrounDRb::MetaWorker
  set_worker_name :billing_worker
  pool_size 1

  def create(args=nil)
    ...
  end
```

```
  def mail_receipts(customer_ids)
    thread_pool.defer(customer_ids) do |customer_ids|
      customer_ids.each do |customer_id|
        CustomerMailer.deliver_receipt(customer_id)
      end
    end
  end
end
```

Now no matter how many mail tasks we send to the worker, they'll all be queued in a thread pool. And since the thread pool has a size of 1, only one of the tasks will run at a time. Similarly, if we wanted concurrent processing of tasks, we could increase the size of the thread pool using the pool_size method and process tasks concurrently.

Last, but by no means least, we might have background tasks that we want to run on an automated schedule. The general solution is to use cron, but managing cron jobs and checking their status can be cumbersome. Thankfully, BackgrounDRb has a built-in scheduler, too. In the backgroundrb.yml file, we can configure certain worker methods to be invoked on a periodic basis (and check it in to version control, of course). For example:

```
---
:backgroundrb:
  :port: 11006
  :ip: 0.0.0.0

:schedules:
  :billing_worker:
    :check_incoming_email:
      :trigger_args: */30 * * * * *
    :generate_reports:
      :trigger_args: 0 30 5 * * * *
      :data: "Summary Report"
```

This configuration schedules the check_incoming_email method of the BillingWorker to execute every 30 seconds and the generate_reports to execute at 5:30 in the morning. The data specified with the :data option is passed to the method as an argument. If you use the scheduler this way, it's important to remember that you should not disable auto loading of worker classes.

Discussion

As of version 1.0, BackgrounDRb is no longer implemented using Ruby's DRb library. And unlike other techniques of network programming that use threads, BackgrounDRb uses IO multiplexing to make use of non-blocking socket IO and stays largely thread free. It does that using the packet[26] library, which is an event-driven network programming library. The basic idea is to have a reactor loop monitoring a socket, and when an event occurs on the socket a callback method is invoked. BackgrounDRb is basically a process that watches for incoming events from Rails and invokes workers to execute Rails code out of band. There are other powerful BackgrounDRb clustering and network programming capabilities available in workers.

Also See

See Recipe 67, *Monitoring (and Repairing) Processes with Monit*, on page 285 for an example of how to configure Monit to monitor the BackgrounDRb server process.

26. http://code.google.com/p/packet/

Part VII

E-mail Recipes

Validating E-mail Addresses

By Michael Slater (http://www.buildingwebapps.com)
Michael Slater is president of Collective Knowledge Works, Inc., which publishes the BuildingWebApps.com portal for Ruby on Rails developers. He's worked as a freelance web developer, director of digital imaging research at Adobe Systems, cofounder and chairman of software start-up Fotiva, editor and publisher of the Microprocessor Report newsletter, hardware engineering consultant, and engineer at Hewlett-Packard.

Problem

When people create an account for your application, they enter their e-mail address. You use that address to send them account activation links, order receipts, and so on. Sometimes people mistype their e-mail address, and they never receive your e-mails. (Instead, you get an e-mail from them complaining they never received thus and such.) So you need to verify their e-mail address before it gets stored in your database.

Solution

We could require that people enter their e-mail address twice in hopes of catching mistakes, but that seems clunky. Or we could send them an e-mail with an activation link which provides the only true validation. Wait, that won't work! We don't have a valid address to send it to.

What we really need to do is validate the e-mail address when the user enters it. There are two parts to this solution: checking that what they enter looks like an e-mail address *and* checking that the domain name is valid. Let's tackle the tricky part first.

The only way to check the domain name is to actually go out over the network and ask the domain if it handles e-mail. We can do that with the help of the Ruby standard resolv library:

Download buffet/app/models/person.rb

```
require 'resolv'

EMAIL_PATTERN = /(\S+)@(\S+)/

SERVER_TIMEOUT = 3 # seconds

def valid_domain?(email)
  domain = email.match(EMAIL_PATTERN)[2]
```

```
  dns = Resolv::DNS.new

  Timeout::timeout(SERVER_TIMEOUT) do

    # Check the MX records
    mx_records =
      dns.getresources(domain, Resolv::DNS::Resource::IN::MX)

    mx_records.sort_by {|mx| mx.preference}.each do |mx|
      a_records = dns.getresources(mx.exchange.to_s,
                                   Resolv::DNS::Resource::IN::A)
      return true if a_records.any?
    end

    # Try a straight A record
    a_records = dns.getresources(domain, Resolv::DNS::Resource::IN::A)
    a_records.any?
  end
rescue Timeout::Error, Errno::ECONNREFUSED
  false
end
```

After teasing out the domain name from the rest of the e-mail address using a regular expression, we use the resolv library to look for MX (mail exchanger) records at the domain. If it finds mail records, they just contain a domain name, so the inner check verifies that the name corresponds to a valid domain. If we find an MX record with a name that matches an A record, then that's the best shot we have at being able to send e-mail to that domain. Except a server doesn't need an MX record to receive an e-mail. So if no MX records were found or had valid domain names, we fall back to checking for an A record. And if doing all this takes too long because a DNS server times out, we treat the e-mail as being invalid.

Finally we combine this with simple formatting validation, add it all to our Person model for example, and we have everything we need:

Download buffet/app/models/person.rb

```
validates_format_of :email, :with => EMAIL_PATTERN

def validate
  unless errors.on(:email)
    unless valid_domain?(email)
      errors.add(:email, 'domain name appears to be incorrect')
    end
  end
end
```

Only if the e-mail matches a loose format do we then go ahead and try to run the network tests. And if everything shakes out, then we wind up with what we believe to be a valid e-mail address in our database.

This is a good step forward: it protects against typos. However, it doesn't validate that there's a working SMTP server living at the other end. We could try to ping the SMTP server through port 25, but modern spam prevention techniques implemented by many servers make this difficult, and potentially very slow.

Discussion

A lot of people try using a fancy, strict regular expression they found on the web to check the e-mail format. While certainly a challenging exercise in regex mojo, these days valid e-mail addresses can have all kinds of weird and wacky characters. I've had better luck using the really basic regular expression shown in this recipe. Your mileage may vary, as they say.

It's possible to take this a step further by sending the SMTP server referenced in the MX record a RCPT TO: command. In theory, this would check that the user name is valid as well as the domain name. However, it takes additional time and mail servers don't always respond reliably.

Also See

You might also want to look at the Email Veracity plugin, [27], which runs similar validations. We chose to implement our own recipe from scratch to demonstrate the concepts.

27. http://rails.savvica.com/2007/11/6/email-veracity-plugin

Receiving E-mail Reliably via POP or IMAP

By Luke Francl (http://railspikes.com)
Luke Francl is a Ruby on Rails developer for Slantwise Design, a Minneapolis-based Rails consultancy. He is a contributor to the MMS2R project and an active member of the Ruby Users of Minnesota. Luke has presented on Ruby and Rails at conferences world-wide. Luke comes to Rails from the Java world and still thinks foreign key constraints belong in the database.

Problem

You need to process incoming e-mail with your Rails application, but you want to make sure you can handle large volumes of e-mail (or a e-mail bomb) without breaking a sweat. You might also not want to configure and run your own mail server.

Ingredients

- An e-mail account that you can access via POP3 or IMAP. If you have a choice, IMAP is better because you can move messages that can't be processed to a folder (call it "bogus" if you want) for later investigation.
- The Fetcher plugin:

  ```
  $ script/plugin install svn://rubyforge.org/var/svn/slantwise/fetcher/trunk
  ```

  The fetcher includes code to download e-mail from POP3 and IMAP servers and includes back-ports from Ruby 1.9 to support secure POP and the PLAIN authentication type for IMAP. It also contains utility code to generate long-running daemon processes, using the Daemon::Base library.[28]
- Optionally, the MMS2R gem:[29]

  ```
  $ gem install MMS2R
  ```

  This is a time-saver for dealing with multipart MIME messages that have attachments. It's targeted at MMS messages, but works for all e-mail with attachments.

28. http://snippets.dzone.com/posts/show/2265
29. http://mms2r.rubyforge.org

Solution

Imagine we're at the helm of a popular web site that lets people e-mail recipes (and a picture of what came out of the oven) to our application, and all the recipes get displayed on page that updates every few minutes.

The most frequently-given solution is to create a procmail rule something like this:

```
:0 c
* ^To:.*@example.com
| /your/rails/app && ruby script/runner "MailProcessor.receive(STDIN.read)"
```

This rule says to take all e-mail sent to any address at example.com and send it to a class called MailProcessor. This approach has two drawbacks for our situation:

1. It will fork a Rails process for each e-mail we receive. We hope to get a lot of e-mail, and this approach may overwhelm our server.
2. It requires that we run an MTA such sendmail or postfix, perhaps even with special mail delivery rules, which (as anyone who's tried to configure one knows) is a task best avoided and left to professionals.

Instead, we're going to use a third-party mail server via POP3 or IMAP, and let them sweat the details. Using POP3 or IMAP to access our e-mail requires a little more code than the procmail route, but it won't hurt a bit.

First, we need an ActionMailer::Base subclass to handle the e-mail when it comes in:

Download ReceivingEmail/app/models/mail_processor.rb

```
class MailProcessor < ActionMailer::Base

  def receive(mail)
    # Your e-mail handling code goes here
    logger.info("Received a message with the subject '#{mail.subject}'")
  end

end
```

Then we need a way to fetch e-mail from the mail server and deliver it into our MailProcessor model. And of course we want it to run continuously so that we get e-mail in a timely manner. The fetcher plugin handles all that for us.

So next we generate a daemon to fetch the mail:

```
$ script/generate fetcher mail
```

In this case, we'll end up with a fetcher called MailFetcherDaemon in script/mail_fetcher. We also get a config/mail.yml configuration file which we need to update with the e-mail accounts we'll use in different Rails environments:

Download ReceivingEmail/config/mail.yml

```
development:
  type: pop
  server: mail.example.com
  username: recipes@example.com
  password: yum

test:
  type: pop
  server: localhost
  username: username
  password: password

production:
  type: pop
  server: localhost
  username: username
  password: password
```

We have a number of configuration options, including:

- type: pop or imap
- server: the IP address or domain name of the server
- port: the port to connect to (defaults to the standard ports)
- ssl: set to any value to use SSL encryption (POP3 only)
- authentication: the authentication scheme to use (IMAP only). Supports LOGIN, CRAM-MD5, and PASSWORD (defaults to PLAIN).
- sleep_time: the number of seconds for the generated daemon to sleep between fetches (defaults to 60 seconds).

Then we just need to edit the generated fetcher daemon to use our MailProcessor class as the receiver of e-mail:

Download ReceivingEmail/script/mail_fetcher

```
def self.start
  puts "Starting MailFetcherDaemon"
  @fetcher = Fetcher.create({:receiver => MailProcessor}.merge(@config))
```

```
  loop do
    @fetcher.fetch
    sleep(@sleep_time)
  end
end
```

The Fetcher.create factory method creates a Fetcher::Imap or Fetcher::Pop instance (depending on your configuration settings) with the MailProcessor as the receiver. When the Fetcher#fetch method is called, the e-mail from the configured mail server is downloaded. Then each message is fed to MailProcessor#receive in turn and deleted from the mail account. In between fetches, the daemon sleeps for the configured sleep time or a default of 60 seconds.

All that's left is to fire up the daemon with the proper Rails environment (it defaults to development):

```
$ RAILS_ENV=production script/mail_fetcher start
```

Once you start the daemon, it keeps running until you call stop. However, in the case of problems or server restart, the daemon won't start up automatically.[30]

Discussion

This solution required some extra code to keep the fetcher running continuously. Why not just use cron for this? Using cron would probably work fine in most cases. In fact, you could use the fetcher plugin for this too:

```
# Run the fetcher every minute with cron
* * * * * script/runner 'Fetcher.create({:receiver => MailProcessor,
  :type => "pop", :server => "mail.example.com",
  :authentication => "PLAIN", :username => "username",
  :password => "password" }).fetch'
```

A drawback to this approach is that if the e-mail takes more than a minute to process, another cron job will start up and process the same e-mail twice. Unexpected results may occur! The daemon will always run for as long as it takes to process the current e-mail in the mailbox, then sleep for :sleep_time seconds. This ensures there's only one process accessing the same e-mail box at a time (assuming you don't start up two daemons!)

30. A good way to resolve this is to use Monit as described in Recipe 67, *Monitoring (and Repairing) Processes with Monit*, on page 285.

As an alternative to using the fetcher plugin, you could roll your own solution using Ruby's built-in support for POP3 (net/pop) and IMAP (net/imap). For example, Benjamin Curtis[31] uses the following script to create Bug model objects in his bug tracker while iterating over a list of e-mails found in an IMAP inbox:

```
require 'rubygems'
require 'lockfile'
require 'net/imap'

Lockfile('lock', :retries => 0) do
  require File.dirname(__FILE__) + '/../config/boot'
  require File.dirname(__FILE__) + '/../config/environment'

  imap = Net::IMAP.new('imap_server_name')
  imap.authenticate('LOGIN', 'imap_login', 'imap_password')

  imap.select('INBOX')
  imap.search(["ALL"]).each do |message_id|
    email = imap.fetch(message_id, 'RFC822')[0].attr['RFC822']
    parsed_mail = TMail::Mail.parse(email)

    unless parsed_mail.to.nil? # Spam
      Bug.create(:tmail => parsed_mail)
    end
    imap.store(message_id, "+FLAGS", [:Deleted])
  end

  imap.expunge
  imap.logout
  imap.disconnect
end
```

Typically this script would be run from cron or daemonized, and it's possible that a run of the script could take long enough to bump into the next invocation of the script. So the script uses the lockfile gem to make sure that only one instance is running at any time.

To keep the details of bug creation out of the IMAP script we create a method in our Bug model that contains the logic for creating a new record given a parsed e-mail (a Mail::TMail object):

```
class Bug < ActiveRecord::Base
  belongs_to :user

  def tmail=(tmail_obj)
    self.user = User.find_or_create_by_email(tmail_obj.from.first)
```

31. http://www.bencurtis.com/

```
            self.summmary = tmail_obj.subject
            self.description = tmail_obj.body
  end
end
```

Keeping E-mail Addresses Up To Date

By Mike Mangino (http://www.elevatedrails.com)
Mike Mangino is the founder of Elevated Rails (http://www.elevatedrails.com). He lives in Chicago with his wife Jen and their two Samoyeds.

Problem

E-mail, for better or worse, remains the primary way to communicate with your application users: sending password resets, account confirmations, order receipts, etc. Unfortunately, e-mail addresses become invalid at a rate of around 15 to 20 percent a year. So how do you keep the e-mail addresses for your users up to date?

Solution

The key to keeping valid e-mail addresses is proactive maintenance. If you regularly correspond with users, detecting bounces can be simple and painless. Indeed, the first edition of *Rails Recipes* [Fow06] contains a recipe on handling e-mail bounces. However, some e-mail servers don't provide bounce messages in a format suitable for that recipe.

We'll tackle how to handle bounced e-mail in two steps: detecting bounces, and notifying e-mail owners of the problem and how to fix it.

The easiest way to identify bounces is to use a consistent reply-to and from address in all the messages we send. Addresses such as no-reply@example.com or bounces@example.com tend to work well. If we receive an e-mail at one of these addresses, it's a strong indication of a bounce (but not always guaranteed).

Once we've received a bounced message, we need to associate it with a user. There are two simple methods we can use, depending upon how much control we have over our e-mail environment. If we have complete freedom in the e-mail addresses we can use, it's easiest to use a unique e-mail address per recipient. For example, if all e-mail sent to the user with id 37 is sent with a from address and a reply-to of no-reply-37@example.com, then we can easily match it up with the right user. One way to do this consistently is with a setup_for_user helper method in our mailer model:

Download EmailBounces/app/models/from_notifier.rb

```
class FromNotifier < ActionMailer::Base

  def new_comment(user, comment)
    setup_for_user(user)
    @subject += " A new comment has been left for you!"
    @body[:comment] = comment
  end

  def setup_for_user(user)
    recipients  user.email
    from  "No Reply <bounces-#{user.id}@example.com>"
    @subject = "[APP_NAME] "
    @body[:user] = user
  end

  def receive(email)
    Bounce.create_for(email)
  end

end
```

If we don't have that sort of flexibility in our e-mail environment, we can instead encode user information in a custom e-mail header. Most e-mail servers will give back the entire bounced message, including headers. We can easily add a custom header to mark which user the bounce came from:

Download EmailBounces/app/models/header_notifier.rb

```
class HeaderNotifier < ActionMailer::Base

  def new_comment(user, comment)
    setup_for_user(user)
    @subject += " A new comment has been left for you!"
    @body[:comment] = comment
  end

  def setup_for_user(user)
    recipients  user.email
    from  "No Reply <bounces@example.com>"
    @subject = "[APP_NAME] "
    headers["X-User-ID"] = user.id
    @body[:user] = user
  end

  def receive(email)
    Bounce.create_for(email)
  end

end
```

In both of our mailers, we included a receive method. (See Recipe 28, *Receiving E-mail Reliably via POP or IMAP*, on page 145 for a recipe on calling this method when an e-mail arrives at your mail account.)

To track bounced e-mails, we'll create a bounces database table, where each bounce belongs to a user:

Download EmailBounces/db/migrate/002_create_bounces.rb

```
def self.up
  create_table :bounces do |t|
    t.integer :user_id
    t.text :body

    t.timestamps
  end
end
```

Then we need a Bounce model to deal with bounced e-mails. It's first job is to parse incoming (defunct) e-mails:

Download EmailBounces/app/models/bounce.rb

```
def self.create_for(email)
  body = email.to_s

  return unless body.match(/MAILER-DAEMON/i)

  email.to.each do |recipient|
    address = recipient.split(/@/)[0]
    if address and match = address.match(/bounces-(\d+)/)
      process_match(match, email)
    end
  end

  if match = email.to_s.match(/X-User-ID:\s+(\d+)/mi)
    process_match(match, email)
  end
end
```

The create_for method first looks for the MAILER-DAEMON string to make sure the message is a bounce. If it isn't found, processing stops. Next, it looks at both the recipients and the headers to try to find a user ID associated with this message. If it finds an e-mail address and a user id, then the pair are handed off to the process_match method:

Download EmailBounces/app/models/bounce.rb

```
def self.process_match(match, email)
  user_id = match[1]
  user = User.find(user_id)
```

```
  cleanup_old_bounces(user)
  bounce = create!(:user => user, :body => email.to_s)
  user.email_bounced(bounce)
end

def self.cleanup_old_bounces(user)
  old = user.bounces.find(:all,:conditions => ["created_at < ?", 21.days.ago])
  old.each(&:destroy)
  user.bounces.reload
end
```

This code creates a new bounce record associated with the user, and purges old bounces to keep things tidy. (Saving off the contents of the message takes a little extra space, but it also makes bounce debugging possible.) Finally we call the email_bounced method on the User model to signal a bounce condition.

How the User object responds to the bounce will depend greatly upon your app's particular usage pattern. Let's assume that e-mail is a crucial component to our app, and we want to correct the e-mail address as quickly as possible. To do that, we'll just set a email_validated_at flag indicating that the user must update or verify their address. For example:

Download EmailBounces/app/models/user.rb

```
class User < ActiveRecord::Base
  has_many :bounces

  MAX_BOUNCES = 1

  def should_email?
    email_validated_at?
  end

  def email_bounced(bounce)
    if bounces.size > MAX_BOUNCES
      update_attribute(:email_validated_at, nil)
    end
  end

end
```

Then over in our ApplicationController we just need two before_filter methods to make sure a user is logged in *and* has a validated e-mail address on file:

Download EmailBounces/app/controllers/application.rb

```
before_filter :login_required
before_filter :require_valid_email
```

```
def login_required
  @user = User.find(session[:user_id])
  redirect_to new_session_path and return false if @user.nil?
end

def require_valid_email
  unless @user.should_email?
    render :action=> "users/validate_email"
    return false
  end
end
```

We'll also need to make sure that all pages a user might visit to change and/or reconfirm their e-mail address skip this before_filter using:

```
skip_before_filter :require_valid_email,
                   :only => [:edit, :verify, :update]
```

Because e-mail does occasionally bounce due to misconfigured servers, you may not want to disable an account on the first bounce. It often makes sense to send a verification e-mail 24 to 48 hours after the first bounce. If that e-mail bounces, it's probably safe to mark the address as bouncing. If that e-mail doesn't bounce, it makes sense to clear the bounce history. You can adjust the MAX_BOUNCES constant used in this recipe to control how many bounces are necessary to disable an account. If you have other methods of communicating with your users, such as RSS or SMS, you may want to use these channels to notify them of e-mail bounces, as well.

Discussion

Bounce messages come from the SMTP server that you use to send e-mail. Be sure to test with your production SMTP server, as implementations vary between vendors.

If you use IDs in your bounce detection that are easy to guess, a malicious user could potentially shut down another users account. This could be a real problem for auction- or finance-related applications. There is more than one solution to prevent this. First, instead of using the ID column, use a UUID or other long and fairly random ID. You could also record each message sent, and use a unique ID per message. This is more secure and can give you better traceability, but may not be worth the trouble, especially if you send a large number of messages.

Part VIII

Console Snacks

Writin' Console Methods

By day, PJ Hyett and Chris Wanstrath run the Rails consulting and training firm, Err Free. By night, they maintain Err the Blog, a running commentary on both Ruby and Rails.

A lot of the irb tricks you develop or find on the web are useful more than once. You know, that obscure snippet that honks your computer's horn whenever a NoMethodError is raised is gonna come in handy for all sorts of fun, so you might as well keep it around.

So where do we stockpile irb goodies? Well, as a courtesy, when irb starts up it tries to load a file named .irbrc from your home directory. And, since script/console is just irb with a tuxedo t-shirt on, all of the irb hacks and customizations are always available. But we can do one better: we can write Rails-specific irb methods and use them across different Rails apps.

Say we routinely run arbitrary SQL from script/console the longhand way, like this:

```
$ ruby script/console
>> ActiveRecord::Base.connection.select_all 'show databases'
=> [{"Database"=>"activerecord_unittest"}, {"Database"=>"err_dev"} ... ]
```

Now let's save some typing by bottling this up in a method. To keep our Rails-specific console methods all together, we'll throw them in a file called .railsrc in our home directory. Here's our new method:

Download console/.railsrc

```
def sql(query)
  ActiveRecord::Base.connection.select_all(query)
end
```

Then we just load up the .railsrc file from within our .irbrc file, which gets loaded when script/console is run. So in .irbrc we have:

Download console/.irbrc

```
if ENV['RAILS_ENV']
  load File.dirname(__FILE__) + '/.railsrc'
end
```

Now we can get at those hard to reach places with ease:

```
$ ruby script/console
>> sql 'show databases'
=> [{"Database"=>"activerecord_unittest"}, {"Database"=>"err_dev"} ... ]
```

Windows Note

On Windows, .irbrc should be kept in C:\Documents and Settings\YourWindowsUsername. The HOME environment variable should then be set to that directory's path.

The console is your friend, and it's also extensible, so extend it!

Console Loggin'

By day, PJ Hyett and Chris Wanstrath run the Rails consulting and training firm, Err Free. By night, they maintain Err the Blog, a running commentary on both Ruby and Rails.

When you're playing around in script/console, it's sometimes helpful to know which database queries are actually being run. No big deal, all we need to do is tell ActiveRecord that instead of using Rails' default logger, it should use a custom logger pointed at STDOUT (your terminal).

These two lines do the trick:

```
ActiveRecord::Base.logger = Logger.new(STDOUT)
ActiveRecord::Base.clear_active_connections!
```

Let's stick 'em in our .railsrc[32] file, and add a couple methods to turn logging on and off:

Download console/.railsrc

```
def loud_logger
  set_logger_to Logger.new(STDOUT)
end

def quiet_logger
  set_logger_to nil
end

def set_logger_to(logger)
  ActiveRecord::Base.logger = logger
  ActiveRecord::Base.clear_active_connections!
end
```

As a matter of interest, there's an alternative way to set the active logger which does not clear the active database connections:

```
def set_logger_to(logger)
  ActiveRecord::Base.connection.instance_variable_set(:@logger, logger)
end
```

Now when we want to sneak a peek at the SQL ActiveRecord is running, we call loud_logger from within script/console:

```
$ ruby script/console
>> User.find(:first)
=> #<User id: 1 ...>
```

32. This is the file we created in Sorbet 30, *Writin' Console Methods*, on page 157

```
>> loud_logger
=> {}
>> User.find(:first)
  User Load (0.000613)   SELECT * FROM users LIMIT 1
=> #<User id: 1 ...>
>> quiet_logger
=> {}
>> User.find(:first)
=> #<User id: 1 ...>
```

That works great if we're only interested in seeing the SQL. But let's say we're pretending to be a casual web surfer by issuing faux-requests to Rails using the app console helper:

```
$ ruby script/console
>> app.get '/people'
=> 200
```

During that request a bunch of stuff happened behind the scenes and got stuck in development.log. So let's go a step further and turn on system-wide logging in the console[33]. Chuck this in your .railsrc file to make the console logger get set up before Rails does its thing:

```
require 'logger'
Object.const_set(:RAILS_DEFAULT_LOGGER, Logger.new(STDOUT))
```

Now everything Rails would normally log to development.log is now logged to our terminal:

```
$ ruby script/console
>> app.get '/people'
Processing PeopleController#index...
  Session ID: BAh7BiIKZmxhc2hJQz...
  Parameters: {"action"=>"index", "controller"=>"people"}
  Person Load (0.000630)   SELECT * FROM `people`
Rendering template within layouts/people
Rendering people/index
Completed in 0.00949 (105 reqs/sec) | Rendering: 0.00180 (18%) |
DB: 0.00063 (6%) | 200 OK [http://www.example.com/people]
=> 200
```

33. Thanks to Tim Lucas for this trick: http://toolmantim.com/article/2007/2/6/system_wide_script_console_logging

Playin' in the Sandbox

By day, PJ Hyett and Chris Wanstrath run the Rails consulting and training firm, Err Free. By night, they maintain Err the Blog, a running commentary on both Ruby and Rails.

The console is great for playing around with your models and controllers. It's not nearly as much fun, though, if you have to worry about goofing something up.

No worries. The --sandbox switch has our back. What it does, surprisingly, is sandbox your data for the duration of your script/console session. Here, let's try to wreck a sand castle:

```
$ ruby script/console --sandbox
Loading development environment in sandbox
Any modifications you make will be rolled back on exit
>> Castle.destroy 1
=> #<Castle id: 1, ...>
>> Castle.find(1)
ActiveRecord::RecordNotFound: Couldn't find Castle with ID=1
>> exit
```

Uh oh. Thankfully the sandbox automatically pushes the UNDO button at the end—it runs everything within a database transaction. So any rows we modify or delete for the duration of the script/console session will be returned to their original state on exit. Let's check:

```
$ ruby script/console
Loading development environment
>> Castle.find(1)
=> #<Castle id: 1, ...>
```

Whew!

Tracking down a hard-to-isolate bug with your before_destroy callback? Playing with the idea of mass-updating data? Examining a copy of production data on your staging database? The --sandbox switch is the way to go. To play it safe, you may want to get in the habit of always running in sandbox mode when poking around your production environment. Then when you really *need* to change production data, you can switch back to the live console.

Accessin' Helpers

By day, PJ Hyett and Chris Wanstrath run the Rails consulting and training firm, Err Free. By night, they maintain Err the Blog, a running commentary on both Ruby and Rails.

By now you've discovered that the console isn't just our friend, it's also a power tool. It slices, it dices, and it even knows how to call view helpers. The appropriately named helper object can be used to play with any of Rails' default helper methods, like this:

```
$ ruby script/console
>> helper.pluralize(3, 'blind mouse')
=> "3 blind mice"
>> helper.submit_tag('Do it!')
=> "<input name=\"commit\" type=\"submit\" value=\"Do it!\" />"
>> helper.visual_effect :blindUp, :post
=> "new Effect.BlindUp(\"post\",{});"
```

This is handy for messin' around with built-in view helpers, but in the old days it had one problem: the helper object didn't know about our app-specific helpers. Hey, but it's a new day. Now in Rails 2.0 the sky has opened.

If we want to call any helper method defined in our ApplicationHelper, the methods are right there:

```
$ ruby script/console
>> helper.some_method_from_application_helper
=> true
```

We run into problems though when trying to use the helper object to play with our other helpers. They're not included by default. But we can fix that. If we want to call any of the methods in our PeopleHelper, for example, we just give the helper object some, er, help:

```
>> helper :people
=> #<Object:0x18ab44c ...>
>> helper.some_method_from_people_helper
=> true
```

Just like that, we can call any of our app-specific view helpers through the helper object. Rinse and repeat for any other helper modules you wanna access.

Shortcuttin' the Console

Once you get addicted, you'll never want to leave the console. Creating little console shortcuts can boost your productivity, and make you look cool when demoing at conferences. Here's a fun one.

Let's say our fingers are worn out from typing Order.find(:first) in the console. Instead, we want to type order(:first). In other words, order(*args) should simply be an alias for Order.find(*args).

Now, we want that to work for *all* models in our current application, generically. It sounds difficult, but it's actually quite easy with a bit of meta-programming:

Download console/.railsrc

```
def define_model_find_shortcuts
  model_files = Dir.glob("app/models/**/*.rb")
  table_names = model_files.map { |f| File.basename(f).split('.')[0..-2].join }
  table_names.each do |table_name|
    Object.instance_eval do
      define_method(table_name) do |*args|
        table_name.camelize.constantize.send(:find, *args)
      end
    end
  end
end
```

Using instance_eval we define a shortcut method for every model in our app, which then simply passes the buck on to the real find find method. We define the shortcuts on the Object class so that script/console can call the methods directly, as if they were built right into the console.

Then we need to make sure our define_model_find_shortcuts method is run when we fire up the console, but *after* all the Rails helpers have been loaded. It turns out irb has a neat configuration option for this:

Download console/.railsrc

```
IRB.conf[:IRB_RC] = Proc.new { define_model_find_shortcuts }
```

The IRB.conf(:IRB_RC) configuration value takes a Proc object, which irb will dutifully run whenever the context is changed. Sometime after Rails loads, the context changes and our shortcuts get defined. Toss all this code in our .railsrc file[34], and we can call the shortcuts with any standard

34. This is the file we created in Sorbet 30, *Writin' Console Methods*, on page 157

finder options:

```
$ ruby script/console
>> order(:first)
=> #<Order id: 1 ...>
>> order(1)
=> #<Order id: 1 ...>
>> order(:all)
=> [#<Order id: 1 ...>]
```

Part IX

Testing

Creating Your Own Rake Test Tasks

Problem

Not all tests are created equal. Some run fast, some slow. Most can be run while you're on an airplane, but some may need a network connection. Of course, all of them should pass all the time. But when you're working on code that has fast, localized tests, you don't want the slow, networked tests to get in your way. Instead, you want to organize the tests around what they need and how frequently they're run.

Solution

Rails provides default Rake tasks for running unit, functional, and integration tests. If we run rake test:units, for example, all the tests in the test/unit directory are run in batch. But what if we have a suite of tests that don't naturally fit into one of the three test buckets? Well, we make a new bucket.

Let's say we have a performance-critical algorithm in our application. It must run within a second or users start shopping around for a new site (and we end up shopping around for a new job). The trouble is, this algorithm is fairly sensitive to changes. So we write a test that uses the Benchmark library to time how long the algorithm takes to run. If we introduce a change that slows it down too much, the test will let out a yelp. Here's what it looks like:

Download buffet/test/performance/perf_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'

require 'benchmark'

class PerfTest < Test::Unit::TestCase

  def test_performance_critical_code
    time = Benchmark.realtime do
      # run code that should not
      # take more than 1.0 seconds
    end
```

```
    assert time <= 1.0
  end

end
```

Now, we could have put this file in the test/unit directory, but we want our unit tests to run as fast as possible. So we put the performance test file in the test/performance directory to keep it away from all the other tests.

Next we need to create a Rake task that runs all the performance tests in batch. Add this task definition to any file with a .rake extension in the lib/tasks directory:

Download buffet/lib/tasks/testing.rake

```
namespace :test do
  Rake::TestTask.new(:performance => "db:test:prepare") do |t|
    t.libs << "test"
    t.pattern = 'test/performance/**/*_test.rb'
    t.verbose = true
  end
  Rake::Task['test:performance'].comment =
    "Run the performance tests in test/performance"
end
```

This code is very similar to how the default Rails testing tasks are created. The only difference is the name of the target and the directory.

So now we can quickly run all the unit tests, and whenever we're messing around with that performance-critical algorithm, we can also explicitly run the performance tests:

```
$ rake test:performance
```

You can create as many of these testing buckets as you need. For example, we might also have a suite of tests for the code we use to validate e-mail addresses. Checking the e-mail domain name requires a good network connection. So let's just add another Rake task for all the network-related tests:

Download buffet/lib/tasks/testing.rake

```
Rake::TestTask.new(:network => "db:test:prepare") do |t|
  t.libs << "test"
  t.pattern = 'test/network/**/*_test.rb'
  t.verbose = true
end
Rake::Task['test:network'].comment =
  "Run the network tests in test/network"
```

So now when we're on an airplane, we can run rake and all the unit and functional tests will pass. Then when we've jacked into the 'net, we can explicitly run the network tests:

```
$ rake test:network
```

Discussion

Of course, you want to run all your tests at least once per day to detect problems before they compound into bigger messes. Creating custom Rake tasks makes it easy to run logical groups of tests on an automated schedule based on when you need feedback. Here's an example build schedule:

- Unit and functional tests run every 5 minutes
- Integration tests run every hour
- Network tests run every 4 hours
- Performance tests run at 2 a.m. daily

If you're not already testing your code on a recurring schedule, check out CruiseControl.rb[35] to get started in minutes.

35. http://cruisecontrolrb.thoughtworks.com/

Testing JavaScript With Selenium

By Marty Haught, Andrew Kappen, Chris Bernard, Greg Hansen (http://www.logicleaf.com/)
Marty, Andrew, Chris and Greg all work together at Logicleaf, an agile software consulting company. On one of their projects, savemycoupons.com, they developed several streamlined testing processes that included a simpler way to integrate Selenium into the daily development cycle.

Problem

Your application uses AJAX for a creamy smooth user experience. Debugging JavaScript can be a royal pain, so you'd rather put some tests in place that will catch JavaScript bugs before they fall through the cracks.

Ingredients

- The Selenium on Rails plugin

```
$ script/plugin install↩
  http://svn.openqa.org/svn/selenium-on-rails/selenium-on-rails
```

- Windows users should also install the win32-open3 gem

```
$ gem install win32-open3
```

Solution

Let's say our application has a login link, that when clicked drops a login form into an empty div. There are many ways to pull off this sort of dynamic page manipulation, but they all rely on JavaScript to do the heavy lifting. Here's what the view looks like:

Download TestingJSWithSelenium/app/views/users/index.html.erb

```
<script type="text/javascript">
function show_form() {
  form_text = '<form action="/users/login">' +
    '<p>User Id: <input name="user_id" type="text"></p>' +
    '<p>Password: <input name="password" type="password"></p>' +
    '<input value="Submit" type="submit"></form>'
       $('form_area').update(form_text)
}
</script>
```

```
<div id="intro">
  <p id="intro_text">
    Do you want to see this page?
    Just click <%= link_to_function("login", "show_form()") %>.
  </p>
</div>

<div id="form_area">
</div>
```

This is all fairly ordinary stuff. The interesting part is figuring out how to test it, and the implementation suggests a few things we need to verify:

1. Rendering the page displays a login link, but no login form
2. Clicking the login link displays the login form
3. Logging in displays the correct dynamic content on the resulting page

We can handle two of these with a good batch of assert_select calls in a functional test:

Download TestingJSWithSelenium/test/functional/users_controller_test.rb

```
def test_index
  get :index
  assert_response :success
  assert_template 'index'
  assert_select "div#intro p#intro_text a[href='#']", 'login'
  assert_select "div#form_area", 1
  # no form inputs in the div yet
  assert_select "div#form_area input[name='user_id']", 0
end

def test_login
  post :login, :user_id => 'Joe', :password => 'password'
  assert_response :success
  assert_template 'home'
  assert_select "h1.heading", "Thank you for logging in, Joe!"
end
```

The test_index method covers the first requirement nicely: When the page is rendered it has a login link, but no login form is visible. If the form_area div were mistakenly commented out, for example, this test would fail. And the third requirement is covered, too: After a login, we see the user name on the resulting page.

So that leaves us with the second requirement: testing the execution of the JavaScript which displays the login form. To do that, we really need an execution environment for JavaScript code.

Selenium[36] gives us such an environment. It embeds a JavaScript-based test engine within a running browser. And as a nice bonus, it lets us do cross-browser testing. That said, Selenium is a general purpose web testing tool and needs a little coaxing to integrate into a Rails application. The selenium-on-rails plugin makes it seamless.

OK, so let's start by generating a stub Selenium test file:

```
$ script/generate selenium login_test.rsel
```

Specifying the .rsel extension lets us write the test using a thin Ruby wrapper for the client-side JavaScript functions[37] Selenium uses to manipulate the browser. We like Ruby.

Then we fill out the file with a mix of assertions and commands to the browser:

Download TestingJSWithSelenium/test/selenium/login_test.rsel

```
open '/users'
assert_text "css=div#intro p#intro_text a[href='#']", "login"
assert_element_present "css=div#form_area"

click "link=login"
# wait for any form text input to appear
wait_for_element_present "css=div#form_area input"
assert_element_present "css=div#form_area input[type='text'][name='user_id']"
assert_element_present "css=div#form_area
  input[type='password'][name='password']"

type "css=div#form_area input[type='text'][name='user_id']", "Joe"
type "css=div#form_area input[type='password'][name='password']", "pass"
click "css=div#form_area input[type='submit']"
wait_for_page_to_load 3000

assert_text "css=h1.heading", "Thank you for logging in, Joe!"
```

The login_test.rsel file looks similar to the functional test we looked through earlier, but we're speaking the RSelenese lingo[38] here. And this time we can run our JavaScript code. The tests clicks the login link and then we wait for any input fields to appear in the form_area div. Then we

36. http://OpenQA.org
37. http://release.openqa.org/Selenium-core/0.8.0/reference.html
38. The RSelenese documentation can be found in vendor/plugins/Selenium-on-rails/doc/index.html

Starting the App on a Mac

If you're a Mac OS X user, one handy configuration setting is start_server. If it's true, then running rake test:acceptance will start your Rails application in test mode before running the Selenium test suite. Currently this setting is not reliable under Windows and you must manually launch the application in test mode before starting the acceptance tests.

Running Tests in the Browser

The selenium-on-rails plugin also lets you manually run specific tests from within the browser. Just launch your Rails application in test mode and browse to http://localhost:3000/selenium. The Selenium IDE FireFox plugin is also worth trying. It lets you (or a non-programmer friend) record a test interactively and export it in a format usable by the selenium-on-rails plugin. See the Selenium website for more details.

can go ahead and test the actual login process by typing a user_id and password and clicking the submit button. By default the click method is asynchronous, so we must call wait_for_page_to_load. (RSelenese also dynamically adds xxx_and_wait commands for each action, so we could just call click_and_wait in a one-liner.) Finally we check for the expected dynamic content on the resulting page.

Now let's see how it all comes together by running the test in the browser. Selenium looks for a config.yml file in the selenium-on-rails plugin directory to tell it which browser(s) to run. Copy the config.yml.example and edit it for your environment. Then start the Rails application in test mode:

```
$ ruby script/server -e test
```

Now run the Selenium tests:

```
$ rake test:acceptance
```

If everything is configured correctly you'll see invisible fingers run through your test script using each configured browser in turn. When everything finishes you should see test successes for each browser.

That's cool, but when a test passes the first time it makes us doubt that it did anything. So it's time to break some JavaScript and see what happens. In index.html.erb, remove the + sign at the end of the line that generates the password input field. This will generate an incomplete form, and it breaks the application in a subtle way. The functional test will still pass, but the Selenium test fails. Even better, the Selenium test catches the missing submit button when we attempt to click it, even though we neglected to pro-actively check for its existence in the test. By implicitly checking the existence of a form control when manipulating it, we can remove the brittle assertion on the view's structure and just simulate the desired interactions.

Now we have a test that fully validates our custom JavaScript in a concise syntax. We've found that testing workflow across several pages is made much easier by Selenium and lends itself to writing a testing language that naturally expresses what a user is really doing instead of focusing on low level functions of the HTML inner workings. Better yet, this allows us to rewrite the implementation without forcing us to rewrite our brittle functional and integration tests.

Discussion

Running Selenium tests generally take a while longer than functional or integration tests. Because of the slow feedback loop, Selenium is not really a good choice for test-driven development. For the same reason, you may not want to rely on Selenium as your only (or primary) testing layer. For best results, and a restful night's sleep, run a nightly Selenium smoke test.

One current drawback to the selenium-on-rails plugin is that it requires that you do not have any open browser windows. If you launch rake:test:acceptance with an open Firefox window, for example, an error pops asking you to close the window. And even when you do, the tests may not run completely. Hopefully this issue will be addressed in a future version of the plugin.

Mocking With a Safety Net

By Kevin Clark (http://glu.ttono.us)
Kevin Clark is a Ruby hacker. He was a founder of SanDiego.rb, an author of Heckle, and blogs at http://glu.ttono.us. He is currently building tools and infrastructure in Ruby at Powerset in San Francisco and writing a book on testing Rails applications for the Pragmatic Programmers.

Problem

You're writing tests for bits of code that are going to hit an external service (or three) and you're using mocks and stubs with Mocha[39] or FlexMock[40] to return canned data. But let's face it, everyone forgets to set up a mock on occasion. So you need to make sure remote calls aren't made on accident while the tests run.

Solution

Let's say we're using the AWS::S3 library[41] to store and fetch files from Amazon's S3 service. In our application we're making calls to the AWS::S3::S3Object class to query S3. During testing, we don't want to make *any* requests to the S3 service. Instead, all calls to S3Object should let us know by raising an exception.

Rails gives us a clever way to solve this with its test/mocks directories. In testing, the test/mocks/test directory is added to the front of $LOAD_PATH. This means that if we create a file called s3.rb in test/mocks/test, and then require 's3' in our application, the s3.rb file in test/mocks/test will be loaded instead of the "real" file.

Suppose we have require 'aws/s3' in our environment.rb file to load the legit S3 library. We need to mimic that path structure in our test/mocks/test directory. Here's what our mock s3.rb file looks like:

Download buffet/test/mocks/test/aws/s3.rb

```
module AWS
  module S3
    class S3Object
      def self.method_missing(action, *args)
```

39. http://mocha.rubyforge.org/
40. http://onestepback.org/software/flexmock/
41. http://amazon.rubyforge.org/

```
        raise "You forgot to stub #{action}!"
      end
    end
  end
end
```

This file is the only version of the S3Object class that gets loaded during testing. And since we haven't defined any methods in this version, all calls to it will end up going to the method_missing method. It just raises an exception, and that's exactly what we want!

Now let's say we have a test that directly (or indirectly) tries to use the S3 service:

Download buffet/test/unit/s3_test.rb

```
def test_s3_call_will_raise_exception
  picture = AWS::S3::S3Object.find 'headshot.jpg', 'photos'
end
```

Running it gives us an error, of course:

```
1) Error: test_s3_call_will_raise_exception(S3Test):
RuntimeError: You forgot to stub find!
...
test/unit/s3_test.rb:7:in `test_s3_call_will_raise_exception'
```

Now we know exactly where we're missing a stub, and we have a failing test that should pass when the stub is in place.

Discussion

Providing an alternate implementation of an entire class is often too heavy handed and you'll want to be more surgical about which methods of a class raise an exception when you forget to mock. In those cases, instead of using the method_missing trick, you probably want to require the original class in your mock version, open the class, and redefine specific methods.

The other thing to note about this example is that we implemented method_missing on the S3Object*class* by defining self.method_missing. In many cases you want to make sure methods on an *instance* aren't called rather than on the class. In those cases you shouldn't define method_missing on self.

Getting Started with BDD

Problem

You've heard about behavior-driven development (BDD) and you want to give it an honest try, but you want to start with a low-ceremony testing library that plays nicely with all your existing Test::Unit tests.

Ingredients

- The latest revision of the Shoulda plugin:

```
$ svn export https://svn.thoughtbot.com/plugins/shoulda/tags/rel-3.0.4
↩
  vendor/plugins/shoulda
```

Solution

The Shoulda plugin[42] is a thin layer of BDD-style syntax on top of Test::Unit that allows us to seamlessly mix it in with our existing Test::Unit tests and tools. Plus we get a handy set of macros for testing Rails apps that keep our test code concise.

In the BDD style, we start by expressing the behavior we want our code to have *before* writing the code. And we do this in small, incremental steps. So let's say we need a model that represents an event, you know, like a party. We start by creating a regular Rails unit test:

Download events/test/unit/event_test.rb

```
class EventTest < Test::Unit::TestCase
end
```

Now it's time to really think about events. What are they? Well, for starters an event should have a number of required attributes, and the name attribute should always be unique. Here's how we express that with Shoulda's validation helpers inside of our unit test file:

Download events/test/unit/event_test.rb

```
class EventTest < Test::Unit::TestCase
  should_require_attributes :name, :description, :image_location,
                            :starts_at, :location, :capacity
```

42. http://thoughtbot.com/projects/shoulda

```
  should_require_unique_attributes :name
end
```

Now let's add some more expectations. The price should be a positive number, of course:

Download events/test/unit/event_test.rb

```
should_only_allow_numeric_values_for :price

should_not_allow_values_for :price, -1.0,
  :message => /must be greater than or equal to 0/
```

Oh, and real people attend events, so an event should have a many-to-many relationship with attendees:

Download events/test/unit/event_test.rb

```
should_have_many :attendees, :through => :registrations
```

Hey, that's a lot of thinking and we haven't even written the code to make our expectations pass. Now, before writing the code, a card-carrying BDDer would first run all the tests and watch them fail. Better yet, they would have done it stepwise: written one assertion, watched it fail, and then written the code. Unfortunately that makes for some fairly tedious reading material. So let's go ahead and create the Event model with all the validations and associations we need:

Download events/app/models/event.rb

```
class Event < ActiveRecord::Base

  validates_presence_of   :name, :description, :image_location,
                          :starts_at, :location, :capacity
  validates_uniqueness_of :name

  validates_numericality_of :price, :greater_than_or_equal_to => 0.0

  has_many :registrations
  has_many :attendees, :through => :registrations,
                       :source => :user
end
```

Then we run the unit test using any of the standard tools and celebrate a small victory of programming!

Depending on how much or little you know about what you want, you might break this test-code cycle down into snappier feedback loops. Personally, I like a steady diet of programmer treats so I would have

done this in two quick steps: one for the validations and one for the associations.

Moving onward, an event should be free when its price is $0. Now we're talking about Event objects, so we'll need one to test against. And we'll need that object for other tests, as well. So we'll use a context to set that up:

Download events/test/unit/event_test.rb

```
context "An event" do

  setup do
    @event = events(:rails_studio)
  end

  should "be free when the price is $0" do
    @event.price = 0
    assert_equal true, @event.free?
  end

  should "not be free when the price isn't $0" do
    @event.price = 1.0
    assert_equal false, @event.free?
  end

end
```

The context block is just a name for the enclosing scenario, if you will. In this context, we're dealing with an Event object which we initialize in the setup block. The should block is a way of creating a test with a meaningful name. In fact, should blocks just create regular Test::Unit test methods behind the scenes. Inside the should blocks we can use any Test::Unit assertions, common assertions that come with Shoulda, or custom assertions we write.

Now to make it pass, we need to add some code to the Event model:

Download events/app/models/event.rb

```
def free?
  self.price == 0
end
```

One fairly unique feature of Shoulda is nested contexts. Say we're now thinking through the behavior of event capacity. We'll need a couple of tests that require an event arranged a certain way. To do that, we just create a new context block within the one we already have, and define a setup block that tweaks the existing @event object a little:

Download events/test/unit/event_test.rb

```
context "An event" do

  setup do
    @event = events(:rails_studio)
  end

  context "with excess capacity" do

    setup do
      @event.capacity = 2
      @event.registrations = [registrations(:fred_for_rails)]
    end

    should "have spaces remaining" do
      assert_equal 1, @event.spaces_remaining
    end

    should "not be sold out" do
      assert_equal false, @event.sold_out?
    end
  end
end
```

What we end up with here is something fairly readable: An event with excess capacity should not be sold out. We're also able to share the cumulative setup blocks which helps remove duplication in our tests. The setup blocks for the contexts are run in order and before each should block. First the @event object is initialized, then its capacity and registrations are assigned, and finally the should block is run. We'll leave the code that makes it pass to you, dear reader.

We can also mix Shoulda tests in with our existing functional tests, and again use Shoulda macros to keep the code concise. Here's how we'd express what a show action should do:

Download events/test/functional/events_controller_test.rb

```
context "showing an event" do
  setup { get :show, :id => events(:rails_studio) }

  should_assign_to :event
  should_respond_with :success
  should_render_template :show
  should_not_set_the_flash
end
```

Shoulda, woulda, coulda. Now you can!

Discussion

There's also a stripped-down gem version[43] for non-Rails projects, which just includes contexts and should blocks.

Also See

test/spec[44] is another lightweight BDD library that maps many of the standard Test::Unit assertions to a 'should'-like syntax.

43. http://shoulda.rubyforge.org
44. http://chneukirchen.org/repos/testspec/

Describing Behaviour from the Outside-In With RSpec

By David Chelimsky (http://blog.davidchelimsky.net/)
David Chelimsky is the lead developer of RSpec and also leads software development at Articulated Man, Inc, in Chicago, IL. Prior to joining Articulated Man, David developed and taught courses in OO Design, Test Driven Development and Refactoring with Object Mentor, Inc. In addition to exploring Ruby, Rails and Behaviour Driven Development, David likes to play guitar and is learning to speak português.

Problem

You've heard a little bit about Behaviour Driven Development and RSpec. You want to see what it's all about, but you don't know how to get started.

Ingredients

- The RSpec[45] plugin:

```
$ script/plugin install ↩
    http://rspec.rubyforge.org/svn/tags/CURRENT/rspec
```

- The RSpec rspec_on_rails plugin:

```
$ script/plugin install ↩
    http://rspec.rubyforge.org/svn/tags/CURRENT/rspec_on_rails
```

RSpec maintains a CURRENT tag, which will always get you the latest release.

Solution

RSpec supports *Behaviour Driven Development*, which is basically Test Driven Development with more natural, behaviour-centric language. That's a mouthful, but it's really quite simple. In BDD we write *Executable Examples* of how an object should behave, and then write the code that makes that object behave correctly. We aim to make each example concise and focused on a single facet of behaviour of the object being *described*.[46] Let's dive right in.

45. http://rspec.rubyforge.org
46. In BDD, we say we are describing the behaviour of an object rather than testing it.

First we need to bootstrap RSpec into our Rails application:

```
$ script/generate rspec
      create  spec
      create  spec/spec_helper.rb
      create  spec/spec.opts
      create  spec/rcov.opts
      create  script/spec_server
      create  script/spec
      create  stories
      create  stories/all.rb
      create  stories/helper.rb
```

RSpec is under constant development, so you may see slightly different output depending on which version you are using. The important pieces for this recipe are the script/spec script, the spec directory, and the spec/spec_helper.rb file.

Then to make sure everything is happy, let's run the examples using the script/spec command, which is installed with the rspec_on_rails plugin:

```
$ ruby script/spec spec

Finished in 0.00995 seconds

0 examples, 0 failures
```

Great, now it's time to start writing some executable examples. We're going to work from the outside-in: describe the behavior we want and then write the code that satisfies the examples. What we want is a list of names and e-mail addresses for people. So let's start from scratch by writing this example in spec/views/people/index.html.erb_spec.rb:

```
require File.join(File.dirname(__FILE__), "..", "..", "spec_helper.rb")

describe "/people/index.html.erb" do

  it "should list all the good people in an unordered list" do
    render '/people/index.html.erb'

    response.should have_tag("ul") do
      with_tag("li") do
        with_tag("div", "First Person")
        with_tag("div", "first@person.com")
      end
      with_tag("li") do
        with_tag("div", "Second Person")
        with_tag("div", "second@person.com")
      end
    end
  end
```

```
end
```

There's quite a bit going on here. Let's take it step by step.

First, the describe method creates an object that's similar to a test case. The it method creates an object that's similar to a test method.

The render method does just what it says: renders the view template. RSpec uses a custom controller to render views based solely on their paths, which is why we need to supply the full path relative to app/views.

The meat of the example is where we express Expectations[47] about the HTML that should be rendered by the view. Expectations are intended to be easy to read. So we're saying that the response should have a ul tag, and inside that tag we should find other tags that list out the people. By way of comparison, we could describe the same behavior using assert_select method that comes with Rails, like so:

```
assert_select("ul") do
  assert_select("li") do
    assert_select("div", "First Person")
    assert_select("div", "first@person.com")
  end
  assert_select("li") do
    assert_select("div", "Second Person")
    assert_select("div", "second@person.com")
  end
end
```

So now with the example written, let's go ahead and run the examples again, this time with the spec task that comes with the plugin:

```
$ rake spec
```

Hrm, we get an ActionController::MissingTemplate error. To resolve that error, we just need to create the template app/views/people/index.html.erb. Let's leave it blank for now and run the examples again. This time we'll use

```
$ rake spec

Expected at least 1 element matching "ul", found 0.
```

Clearly we need some structure in our template. So let's add the following to app/views/people/index.html.erb:

```
<ul>
<% for person in @people -%>
```

47. In BDD, we refer to Expectations instead of Assertions.

```
  <li>
    <div><%= h person.full_name %></div>
    <div><%= h person.email %></div>
  </li>
<% end -%>
</ul>
```

This time when we run the examples we get "You have a nil object when you didn't expect it!" from our @people instance variable. Well, of course! There are no people. We have a view example and a view template, but there are absolutely no models or controllers.

Let's supply our view with the data that it needs, but let's do it without building out the other pieces. The view wants an array of @people, so let's just simulate one:

Download RSpec/spec/views/people/index.html.erb_spec.rb

```
require File.join(File.dirname(__FILE__), "..", "..", "spec_helper.rb")

describe "/people/index.html.erb" do

  it "should list all the good people in an unordered list" do
    assigns[:people] = [
      stub("person1", :full_name => "First Person",
                      :email => "first@person.com"),
      stub("person2", :full_name => "Second Person",
                      :email => "second@person.com")
    ]

    render '/people/index.html.erb'

    response.should have_tag("ul") do
      with_tag("li") do
        with_tag("div", "First Person")
        with_tag("div", "first@person.com")
      end
      with_tag("li") do
        with_tag("div", "Second Person")
        with_tag("div", "second@person.com")
      end
    end
  end
end
```

The assigns method lets us specify instance variables that will be available to the view. In this case there will be a @people instance variable containing an array of two stub objects.[48] Each stub object returns stub

48. RSpec comes with a built in mocking and stubbing framework. If you are already

(fake) values for full_name and email.

Now let's come full circle by running the examples again:

```
$ rake spec
.

Finished in 0.081351 seconds

1 example, 0 failures
```

As a bonus, let's list out all of our expectations:

```
$ ruby script/spec spec --format specdoc

/people/index.html.erb
- should list all the good people in an unordered list

Finished in 0.113134 seconds

1 example, 0 failures
```

And while we're at it, let's also generate a nice HTML report:

```
$ ruby script/spec spec --format html:rspec_report.html
```

All the things we expect our object to do are listed in the rspec_report.html file.

That's all there is to it—we've described our first bit of Rails behaviour using RSpec. We started with a failing Executable Example, added the code to make it pass, and did so without relying on the existence of any controllers or models in our app.

In addition, without even thinking about it, we've discovered exactly what our controller will need to provide for the view (a collection of people) and that each person will need to have first_name and last_name attributes. Imagine how powerful that can be when you're dealing with a complex model. This is what *Outside-In* is all about: we start with the outermost layers and let them guide us all the way down to the low level components of our applications.

---

familiar with either FlexMock or Mocha, you can use those instead.

Discussion

We went through that example pretty quickly, but there are a few interesting things that happened that we should talk about.

RSpec supports a philosophy that you should be able to describe each component in isolation from each other. For this to work effectively, you should include some level of integration testing [49] in your process. If you don't, it's possible to get all of the objects working correctly in isolation and then watch things fall apart when you fire up your application.

If you're experienced with TDD or BDD, you probably recognized that starting with so much of the example was a bigger step than we normally take in practice. Doing TDD/BDD with discipline, we would start with a much more granular step: perhaps a single line stating an expectation that some specific text is rendered. Then we follow the errors until the example passes, and then add more detail to the example, follow the errors, get it to pass, rinse, repeat, until the passing example expresses the detail that we see above.

Did you notice that we wrote the code in the view before the stubs? Did that strike you as backwards? Well, it is backwards from how a lot of people use stubs, but think about this: we started by expressing our desired outcome, the list of people. That desired outcome led us to write the code that we expect produce that outcome. It was only then, after we wrote that code and could see it, that we knew exactly what stubs to write. In the same way that the example expressed expectations of the view, the view code expressed expectations of its environment, which we then satisfied with the stubs.

Another thing you might have noticed is that the have_tag expectation is very specific. It will fail unless the particular tags are present with the correct nesting structure. This much detail tends to make the examples quite brittle, meaning that the example needs to change every time the HTML design changes.

Generally, the approach I take is to put in only the detail that has relevant business value. For example, a form won't work correctly if the input elements are not nested inside the form element. For that reason I'll usually be specific about the structure of a form. Another case might

49. RSpec >= 1.1.0 supports integration testing with the Story Runner, including a wrapper for Rails integration tests.

be that there is javascript and/or CSS that will fall apart if things aren't structured in a specific way. Again, depending on the business value of that javascript or CSS, I might include that detail.

RSpec does express some opinions. In this example, the fact that we could render a view with no underlying controller and model is quite frightening to some who aren't experienced with this style of testing. Rails' functional tests express a different opinion, that you should run all the pieces in each of your tests to make sure they all work together. If you prefer that approach, you *can* achieve this with RSpec using RSpec's Controller Examples in Integration Mode.

Reducing Dependencies with Mocks

By Matthew Bass (http://matthewbass.com)
Matthew Bass is an independent software developer who has been enjoying the freedom of Ruby for many years now. He is a speaker, agile evangelist, and Mac addict. He co-organizes the Ruby Meetup in his home town of Raleigh, North Carolina and blogs at matthewbass.com.

Problem

Your functional tests always seem to be breaking, and for the wrong reasons. It's usually because someone changed an implementation detail down in a model. You need to reduce the dependencies your controllers have on the rest of the system, but how do you do it without adding a bunch of special testing hooks?

Ingredients

- The FlexMock gem[50]:

```
$ gem install flexmock
```

Solution

Say we have a model with a wee bit of validation:

Download MockingCornerCases/app/models/gadget.rb

```
class Gadget < ActiveRecord::Base
  validates_presence_of :name, :price
end
```

And we also have a typical controller action for creating a gadget, something like this:

Download MockingCornerCases/app/controllers/gadgets_controller.rb

```
def create
  @gadget = Gadget.new(params[:gadget])

  if @gadget.save
    flash[:notice] = "Gadget was successfully created."
```

50. http://onestepback.org/software/flexmock/

```
    redirect_to @gadget
  else
    flash[:error] = "Whoops, gadget couldn't be created."
    render :action => "new"
  end
end
```

Being good little programmers, we have a functional test for the action:

Download MockingCornerCases/test/functional/gadgets_controller_test.rb

```
def test_should_create_gadget
  assert_difference('Gadget.count') do
    post :create, :gadget => { :name => "Chronometer", :price => 6 }
  end
  assert_not_nil assigns(:gadget)
  assert_not_nil flash[:notice]
  assert_redirected_to gadget_path(assigns(:gadget))
end
```

However, we're only verifying that the action does the right thing if the gadget can be saved. We don't have a test for what happens if the gadget cannot be saved. And running rcov tells us that we're pretty bad about testing corner cases like this in general. Hmph.

Now, to test the failure case we could pass an empty hash to the create action. Since our model is validating the presence of the name and price attributes, the save would fail.

But that would make our test a tad brittle. If we were to remove the gadget validations in future, our failure test case would fail. In other words, our test is too tightly coupled to the presence of validations in our model. Instead, our test should be verifying one thing and one thing only: that if the gadget fails to save, the controller handles that failure correctly.

So let's simulate a failure by introducing a mock object into the mix. First we need to require FlexMock's Test::Unit helper inside our test/test_helper.rb file:

```
require "flexmock/test_unit"
```

Next, we need to mock the save method on new instances of our Gadget class. The save method should return false so it triggers our failure condition:

Download MockingCornerCases/test/functional/gadgets_controller_test.rb

```
def test_create_invalid_gadget_fails
  flexmock(Gadget).new_instances.should_receive(:save).
                   once.and_return(false)
```

```
  post :create, :gadget => { }
  assert_not_nil assigns(:gadget)
  assert_response :success
  assert_template 'new'
  assert_not_nil flash[:error]
end
```

This test won't break if we change validations on the Gadget model. In fact, the save fails regardless of the parameters we POST to the create action. And that's exactly what we want. We aren't concerned with the internals of the Gadget model or whether or not the save method is actually working. We have unit tests for that. In our functional test, we're only concerned with how our controller responds when save returns false. The use of a mock lets us precisely control our gadgets so they behave in a predictable way.

This is all well and good, but after writing failure tests for several controllers we begin smelling duplication:

- Try to save the model.
- If the save succeeds, populate the flash and redirect.
- If the save fails, populate the flash and fall through to the default render.

Duplication in tests is just as bad (if not worse) as duplication in production code. So let's clean this up with some meta-programming that builds failure tests on the fly:

Download MockingCornerCases/test/test_helper.rb

```
def self.test_create_invalid_fails(options={})

  if options[:model]
    model = options[:model]
  else name.demodulize.to_s =~ /^(.*)ControllerTest$/
    model = $1.singularize.constantize rescue nil
  end

  define_method("test_create_invalid_fails") do
    flexmock(model).new_instances.should_receive(:save).
                    once.and_return(false)
    post :create
    assert_not_nil assigns(model.to_s.underscore)
    assert_response :success
    assert_template 'new'
    assert_not_nil flash[:error]
  end
end
```

If a model class name is passed in via options then we use it. Otherwise, we try to glean the model class name from the functional test name. Then we dynamically create a test method that does exactly what we did before, but in a generic way. So now in our functional test, we can just write:

Download MockingCornerCases/test/functional/gadgets_controller_test.rb

```
test_create_invalid_fails
```

And we could do the same thing with the happy path, replacing our original successful creation test with a one-liner.

Discussion

This kind of automation can be quite useful for testing a large system with many controllers that do similar things. RESTful controllers, in general, are ideal candidates for meta-programming.

Also See

- This is the mere tippy-top of the FlexMock iceberg. It has an extensive (and yet extremely usable) mocking API, so be sure to check out the excellent documentation for all your mocking needs.

Fixtures Without Frustration

Problem

Text fixtures have become a millstone around your neck, dragging you down every time you try to be a good little tester. You spend hours getting all the fixture records knitted together with ids, only to have it all come crumbling down with the slightest change to test data. If only you didn't have to remember all those numbers, life would be a little sweeter (and you'd actually get some tests written).

Solution

Let's start with a little recap of just how quickly text fixtures can go bad. Suppose we have a has_and_belongs_to_many relationship between users and tags. To populate the tags_users join table of our test database, we painstakingly created the following tags_users.yml fixture file:

```
fred_caveman:
  user_id: 1
  tag_id: 1

fred_programmer:
  user_id: 1
  tag_id: 3

barney_caveman:
  user_id: 2
  tag_id: 1

barney_juggler:
  user_id: 2
  tag_id: 2
```

Tying these relationships together with primary keys really hurts, and it leads to brittle fixture files. Oh, and it makes the fixture files hard to decipher, too: What exactly are Barney's tags? It's pretty much all bad news. So let's clean this mess up using the new (foxy) fixtures in Rails 2.0.

First, we delete the tags_users.yml fixture file altogether. We don't need it because we already have a tags.yml file, and each tag record has a label:

Download events/test/fixtures/tags.yml

```
programmer:
```

```
  name: Programmer

juggler:
  name: Juggler

caveman:
  name: Caveman
```

Then we update the users.yml fixture file and tag each user using the labels from our tags.yml file:

Download events/test/fixtures/users.yml

```
fred:
  name: Fred Flintstone
  email: fred@flintstones.com
  tags: caveman, programmer
barney:
  name: Barney Rubble
  email: barney@rubbles.com
  tags: caveman, juggler
```

Now that's more like it! Rather than using ids, we can just use the tags association because the fixture knows that our User model declares a has_and_belongs_to_many association to our Tag model.

In fact, we don't need to type *any* ids into our fixture files—they're generated for us by hashing the fixture record label. All we need to remember is the label. And if we really need the id for a label, for example when patching up old fixtures, we can turn it inside out using ERB in our fixture file:

```
tag_id: <%= Fixtures.identify(:caveman) %>
```

Or we can track down ids from Rake:

```
$ rake db:fixtures:identify LABEL=caveman
```

We can also use fixture labels to clean up another association in our app: an event has_many users through registrations, and vice versa. That is, we have a join model called Registration that points to both an Event and a User. In this case, because a Registration isn't a pure join table, we need a registrations.yml fixture file for the registrations table. In the old days the fixture file would be littered with foreign_keys, like this:

```
fred_for_rails:
  event_id: 1
  user_id: 1
  price: 10.00
  paid_at: <%= 1.day.ago.to_s(:db) %>
```

```
barney_for_ruby:
  event_id: 2
  user_id: 2
  price: 20.00
  paid_at:
```

But now things become a lot more readable:

Download events/test/fixtures/registrations.yml

```
DEFAULTS: &DEFAULTS
  price: 10.00
  paid_at: <%= 1.day.ago.to_s(:db) %>

fred_for_rails:
  user: fred
  event: rails_studio
  <<: *DEFAULTS

barney_for_ruby:
  user: barney
  event: ruby_studio
  <<: *DEFAULTS
```

Again, the fixture knows about the belong_to associations in our Registration model, so we can replace event_id with event, for example. While we're at it, refactoring all the duplicated fixture keys into a set of defaults that each record references makes the fixture even easier to maintain. Any fixture labeled DEFAULTS is ignored, and YAML takes care of the rest.

Things don't get out of sync...

Discussion

Yes, foxy fixtures also support polymorphic associations. For example, suppose we have an Address model that can belong to an Event, a User, or any other model that's *addressable*, like so:

```
class Addressable
  belongs_to :addressable, :polymorphic => true
end
```

Instead of using the id and the type in the fixture file, we can just use the polymorphic target label and type:

```
rails_in_denver:
  address: 10345 Park Meadows Drive
  city: Denver
  state: CO
  addressable: rails_studio (Event)
```

```
fred_in_denver:
  address: 123 Main Street
  city: Denver
  state: CO
  addressable: fred (User)
```

Tracking Test Coverage with RCov

Problem

Writing tests is all well and good, but how do you know when your application is sufficiently tested? And if you're learning to write automated tests, where do you dig in? Well, if your bug tracking system is empty and your phone isn't ringing at 2am, that's a good start. But when the phone does start ringing, it's usually too late. You'd like to get some insight into how well your code is tested *before* it's rolled into production.

Ingredients

- The rcov gem:

```
$ gem install rcov
```

- The rails_rcov plugin (optional):

```
$ script/plugin install http://svn.codahale.com/rails_rcov
```

Solution

There are many different metrics for tracking the efficiency of our tests. One of the easiest metrics to collect is code coverage. Simply put, it tells us which parts of our code get exercised by our tests.

Test coverage is a no-brainer to measure because rcov automates the entire process. We just run our tests—or put them on an automated run cycle—and rcov tallies up statistics. We'll get to the how in a jiffy. First a peek at the HTML output:

C0 code coverage information

Generated on Wed Dec 12 18:07:56 -0700 2007 with rcov 0.8.1.2

Name	Total lines	Lines of code	Total coverage	Code coverage
TOTAL	322	227	88.2%	83.3%
app/helpers/application_helper.rb	25	18	68.0%	55.6%
app/controllers/registrations_controller.rb	13	11	76.9%	72.7%
app/controllers/application.rb	33	19	84.8%	73.7%
app/controllers/events_controller.rb	86	60	86.0%	80.0%
app/controllers/users_controller.rb	37	32	83.8%	81.2%
app/models/user.rb	70	48	94.3%	91.7%
app/models/event.rb	45	30	100.0%	100.0%
app/models/tag.rb	5	3	100.0%	100.0%
app/models/registration.rb	6	4	100.0%	100.0%
app/helpers/events_helper.rb	2	2	100.0%	100.0%

This is telling. Our RegistrationsController only has around 77% test coverage and it plays a central role in our business. That's unfortunate. To see which lines are untested, we just click on the file name:

```
1 class RegistrationsController < ApplicationController
2
3   def create
4     @event = Event.find(params[:event_id])
5     if current_user.register_for(@event)
6       flash[:notice] = "Thanks for registering!"
7     else
8       flash[:notice] = "You're already registered for that event!"
9     end
10    @events = current_user.events
11  end
12
13 end
```

Whoops! We tested the blue-sky scenario, but forgot about the edge case. It's embarrassing, but it's exactly the kind of quick feedback we need to guide our testing efforts. Plus it just feels good to write a new test that pushes the coverage bar to 100%.

That's a taste of the treat; now for the ingredients. All we need is a Rake task that calls the rcov command-line utility and remembers all of our options. We'll stick that in the lib/tasks/rcov.rake file,[51] for example:

51. Thanks to Chris Noble for working out the Windows-specific bits.

Download events/lib/tasks/rcov.rake

```
namespace :test do

  desc 'Tracks test coverage with rcov'
  task :coverage do
    unless PLATFORM['i386-mswin32']
      rm_f "coverage"
      rm_f "coverage.data"
      rcov = "rcov --sort coverage --rails --aggregate coverage.data " +
             "--text-summary -Ilib -T -x gems/*,rcov*"
      system("#{rcov} --no-html test/unit/*_test.rb")
      system("#{rcov} --no-html test/functional/*_test.rb")
      system("#{rcov} --html test/integration/*_test.rb")
      system("open coverage/index.html") if PLATFORM['darwin']
    else
      rm_f "coverage"
      rm_f "coverage.data"
      rcov = "rcov.cmd --sort coverage --rails --aggregate coverage.data " +
             "--text-summary -Ilib -T"
      system("#{rcov} --no-html test/unit/*_test.rb")
      system("#{rcov} --no-html test/functional/*_test.rb")
      system("#{rcov} --html test/integration/*_test.rb")
      system("\"C:/Program Files/Mozilla Firefox/firefox.exe\" " +
             "coverage/index.html")
    end
  end

end
```

Then it's just one command to run all our tests, generate the aggregated HTML report, and pop it open in our default browser:

```
$ rake test:coverage
```

We also get a text summary for the unit, functional, and integration tests as they're being run. Either way, the reports can be quickly scanned for warning signs.

Discussion

We have to say this, otherwise the testing gurus come out of the woodwork: Code coverage is an inexpensive metric to collect, but it should *not* be the only yardstick by which we evaluate our tests. All rcov does is check whether a line of code was executed, not that it's the correct code. Having gotten that out of the way, using rcov is a lot better than throwing arbitrary tests at your application and hoping it's time well spent.

So is it 100% code coverage or bust? Well, that's a worthy goal but sometimes not practical. In some cases rcov can't accurately measure one-liners, for example. Chad Fowler's rule is: "Everything green except when rcov is too dumb to understand that my code is covered". RCov should only be used as a directional indicator. It's a really good one but taking it too literally is a mistake.

Also See

If you're the sort who doesn't mind adding yet another plugin, the rcov_rails plugin gives you Rake task out of the box. For every test:xxx task in your Rails application, the rails_rcov plugin adds two more: test:xxx:rcov and test:xxx:clobber_rcov. For example, to track the coverage of our unit tests, and then remove the unit test coverage reports, use:

```
$ rake test:units:rcov
$ rake test:units:clobber_rcov
```

Testing HTML Validity

By Peter Marklund (http://marklunds.com)
Peter has extensive experience with and expertise in object orientation, web development, relational databases, and testing, and has been doing web development with Java and Tcl since 2000. In late 2004 he was introduced to Ruby on Rails and has since helped develop an online community, and a CRM system with Rails. Peter is currently working as a Ruby on Rails developer and teacher in Stockholm and is helping organize events for the local Rails community.

Problem

You want to make sure your site renders consistently (and correctly) across all modern browsers. A good start is to validate your markup so that it complies with the W3C standards, doesn't have loop holes in HTML escaping, and doesn't contain broken links, form POSTs, or redirects. But how do you automatically validate all that in an application that dynamically generates HTML?

Ingredients

- The Html Test plugin:

```
$ script/plugin install http://htmltest.googlecode.com/svn/trunk/html_test
```

- The RailsTidy plugin[52] is optional, but a useful complement to the W3C validator since it generates warnings about empty tags, for example. RailsTidy depends on the HTML Tidy library as well as its Ruby API.

Solution

The Http Test plugin gives us a handful of assertion methods that we can use in functional and integration tests. Here, let's ask the W3C if our index template passes muster:

```
def test_index
  get :index
  assert_response :success
  assert_template 'index'
  assert_validates
```

52. http://www.cosinux.org/~dam/projects/rails-tidy/doc/

The assert_validates method simply calls two underlying assertions: assert_w3c and assert_tidy. (You'll just call assert_w3c if you aren't using Tidy.) The last generated W3C error report is written to /tmp/w3c_last_response.html for debugging fun.

Those assertions are run after the request is processed, and they're useful if we're only looking to validate a subset of our pages. If instead we want to validate our entire site, we can add this to our test_helper.rb file:

```
ApplicationController.validate_all = true
ApplicationController.validators = [:tidy, :w3c]
```

To make sure that URLs in links, forms, and redirects resolve, we need to add the following to test_helper.rb for all tests or to a specific controller/integration test:

```
ApplicationController.check_urls = true
ApplicationController.check_redirects = true
```

That's pretty handy, but it comes at a price. By default the Html Test plugin uses the online W3C validator.[53] This can significantly slow our tests down (plus we like to run tests while on airplanes). A better solution is to install the W3C validator locally.[54] Then we just point the assertions to our local install by adding this to our test_helper.rb file, for example:

```
Html::Test::Validator.w3c_url = "http://localhost/cgi-bin/check"
```

Now let's say we want to restrict HTML validation to a subset of our controllers or actions. For example, suppose we don't want to run it on the admin side of our application. To do that, we override the should_validate? method to turn off validation under /admin:

```
module Html
  module Test
    class ValidateFilter
      def should_validate?
        response.headers['Status'] =~ /200/ &&
        params[:controller] !~ /^admin/
      end
    end
  end
end
```

53. http://validator.w3.org
54. See http://validator.w3.org/docs/install.html for installation instructions for your operating system.

In a similar fashion, we can override the skip_url? method to skip checking whether a URL in a link, form, or redirect resolves. Here's the default implementation of that method:

```
module Html
  module Test
    module UrlSelector
      def skip_url?(url)
        return true if url.blank? or (!external_http?(url) and url =~ /:\/\//)
        # Unsupported protocols
        [/^javascript:/, /^mailto:/, /^\#$/].each do |pattern|
          return true if url =~ pattern
        end
        false
      end
    end
  end
end
```

Finally, here's a trick for finding unquoted data in your templates: Add characters such as "&" and "<" to your fixture data. That way, HTML validation will fail if unquoted fixture data is included in a template. This of course assumes that the template is covered by a controller or integration test.

Discussion

Tidy can sometimes be a bit too picky. If it barks about things you don't care about, you can tell Tidy to ignore them:

```
Html::Test::Validator.tidy_ignore_list =
  [/<table> lacks "summary" attribute/]
```

Also See

There are a couple of alternative Rails plugins that you might want to look into for your validation needs.

- Assert Valid Markup[55] caches the results and only hits the validator when the generated HTML response changes.
- The RaiLint plugin[56] can validate all HTML, CSS, and JavaScript assuming you have Java, the JavaScript Lint validator, and a few other things installed locally.

55. http://redgreenblu.com/svn/projects/assert_valid_markup/README
56. http://rubyforge.org/projects/railint/

If you have problems installing the W3C validator locally you can use the xmllint command line validator instead. The xmllint validator is part of the libxml2 library that you can download from http://xmlsoft.org. The W3C validator is favored over xmllint because it's more authoritative, and it also produces much nicer and easier to debug error reports with references to line numbers in your document.

Part X

Performance and Scalability Recipes

Looking Up Constant Data

Problem

Your application has constants stored in the database: states, countries, planets, etc. You want to list the constants in a form selection, for example, but you don't want to fetch the constants from the database every time the form is displayed.

Solution

Let's take the example where a person needs to select their home state as part of their profile. We have an empty Person model and a bare-bones migration file that looks like this:

Download buffet/db/migrate/001_create_people.rb

```
class CreatePeople < ActiveRecord::Migration

  def self.up
    create_table :people do |t|
      t.string :name
      t.string :email
      t.string :state
    end
  end

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

On the profile form we want to show full state names in the selection, but when a person saves their profile we want the people.state column to contain the state's abbreviation. So we need a migration file to create the states table where we'll store the state names and abbreviations. When the migration is applied, we'll go ahead and populate the table with all 50 states.

Download buffet/db/migrate/002_create_states.rb

```
class CreateStates < ActiveRecord::Migration

  def self.up
    create_table :states do |t|
      t.string :name
```

```
      t.string :abbreviation
    end

    State.create([
      {:name => 'Alaska',   :abbreviation => 'AK'},
      {:name => 'Alabama',  :abbreviation => 'AL'},
      {:name => 'Arkansas', :abbreviation => 'AR'},
      {:name => 'Arizona',  :abbreviation => 'AZ'},
      # ... more states ...
      {:name => 'Wyoming',  :abbreviation => 'WY'}
    ])
  end

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

Once we have those constants tucked away in the database, we need to read them into our application's memory... exactly once! It turns out Ruby makes that easy to do. When you define a class in Ruby, it's executable code. That means you can have arbitrary code run *while* a class is being defined. And in production mode Rails doesn't reload classes, so they'll be defined one time.

We just need a class to trigger all this, and it seems reasonable to have a State model class for our states table. While the State class is being defined, we can call any method on the State class object. If that hurts your brain, don't worry. In this case it means that we can call find to read in all the state data. Here's what that looks like:

Download buffet/app/models/state.rb

```
class State < ActiveRecord::Base

  NAMES_ABBREVIATIONS = self.find(:all, :order => 'name').map do |s|
    [s.name, s.abbreviation]
  end

end
```

Remember that we want to display the state names, but store a state abbreviation in the person's record. In other words, we don't need to carry around all the attributes of each State model object. Instead, we use the map method to turn each model into an array containing just the name and abbreviation. Let's use script/console to see how the constant data get packaged:

```
$ ruby script/console
```

```
>> State::NAMES_ABBREVIATIONS
=> [["Alabama", "AL"], ["Alaska", "AK"], ["Arizona", "AZ"],
    ["Arkansas", "AR"], ... ["Wyoming", "WY"]]
```

It's just an array of arrays. Now we have a really simple, but effective, "compile-time" cache. From anywhere in our application we can reference State::NAMES_ABBREVIATIONS and get all the state names and abbreviations without hitting the database. Which brings us to the form:

Download buffet/app/views/people/_form.rhtml

```
<% form_for(@person) do |f| -%>
  <p>
    <strong>Name</strong>:
    <%= f.text_field :name %>
  </p>
  <p>
    <strong>Email</strong>:
    <%= f.text_field :email %>
  </p>
  <p>
    <strong>State</strong>:
    <%= f.select(:state, State::NAMES_ABBREVIATIONS) %>
  </p>
  <p>
    <%= f.submit 'Save' %>
  </p>
<% end -%>
```

The select form helper will happily use the array of arrays contained in our State::NAMES_ABBREVIATIONS. (Don't you love it when a plan comes together?) The first element of each array (the state name) is used as the text displayed for the option and the second element (the state abbreviation) is used as the value that gets sent to the server when the form is submitted. Here's the HTML that gets generated:

```
<select id="person_state" name="person[state]">
  <option value="AL">Alabama</option>
  <option value="AK">Alaska</option>
  <option value="AZ">Arizona</option>
  <option value="AR">Arkansas</option>
  . . .
  <option value="WY">Wyoming</option>
</select>
```

That's all there is to it! Works a treat for constant (read-only) data that you want to look up once.

Discussion

Bear in mind that you won't see the benefits of caching unless you're running the application in production mode. In development mode your application's code is reloaded on every request. To test out the caching in development, you need to make the following change (and remember to reset!) in your config/environments/development.rb file:

```
config.cache_classes = true
```

Serving Page Caches to Facebook

By Ezra Zygmuntowicz (http://brainspl.at)
Ezra Zygmuntowicz is a co-founder of EngineYard.com, a scalable Rails hosting service. He is the author of the book *Rails Deployment* by the Pragmatic Programmers and has contributed many open source Ruby and Rails related projects such as BackgrounDrb, ez-where, and Merb. He is a speaker at The Rails Edge, the 2006 and 2007 RailsConf, and the 2007 SDForum Ruby conference. He has been working with Ruby for almost 4 years and picked up Rails in the summer of 2004. In his spare time he likes to hack Ruby and Erlang and tinker with his vintage 54 VW beetle.

Problem

You have a Facebook app and you're using Rails' built-in page caching to serve the page contents directly from disk. Unfortunately, Facebook only sends POST requests and nginx (and most other web servers) don't allow serving a static file in response to a POST request. Nginx, for example, returns a 405 error.

Ingredients

- The nginx web server

Solution

To solve this we need to handle the 405 error and serve up the page-cached file while at the same time handling all the normal requests. Here's the snippet we need to add at the bottom of our vhost server block in our nginx configuration file:

```
error_page   405 =200 @405;
location @405 {
  index  index.html index.htm;
  # needed to forward user's IP address to rails
  proxy_set_header  X-Real-IP  $remote_addr;
  proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header Host $http_host;
  proxy_redirect false;
  proxy_max_temp_file_size 0;
  proxy_next_upstream error;
  if (-f $request_filename) {
    break;
```

```
  }
  if (-f $request_filename.html) {
    rewrite (.*) $1.html break;
  }
  if (!-f $request_filename) {
    proxy_pass http://mongrel;
    break;
  }
}
```

So first we catch the 405 error with an error_page directive and change it to a 200 response code. Then after setting some proxy directives, we check to see if there's a page-cached file that matches the request. If so, we serve it straight from disk and Facebook is happy.

So far, so good. If there's no cached page file, then we do the standard stuff. First we handle requests for static ..html files and fall back to our Rails app via Mongrel if it's a dynamic request.

Tricks like this seem obvious once you get them working and step back, but this one in particular took forever to figure out. Here's saving you the trouble.

Profiling In The Browser

By Aaron Batalion (http://blog.batalion.com)
Aaron Batalion likes long walks in the park and developing web applications. He is an early adopter who picked up Ruby On Rails in 2005 while working at Blockbuster, where he led the architectural effort to build an online subscription portal, from online experience through fulfillment. Later, as an architect for Revolution Health, he led the organizational shift from Java and other technologies to a Rails platform and has enjoyed stretching Rails to its enterprise limits ever since. He has authored and co-authored many plugins, many of which are available on RubyForge.

Problem

As more and more users flock to your application, performance becomes increasingly important. You may have tried profiling certain pages by writing integration tests that are wrapped in timing code. But your Rails app behaves differently when its nestled in its production environment. So you really need to profile your application *in* production to answer questions such as these:

- Is that Amazon API call that you mocked out locally taking 80% of the request time in production?
- Do you have an ActiveRecord finder that was speedy with a small dataset, but needs some love with bigger datasets?
- Is your caching solution paying off as expected?

Ingredients

- The ruby-prof profiler gem:

```
$ gem install ruby-prof
```

Solution

It turns out we can arrange things so that we get profiling information for any web request right in our browser!

First we need a good profiler library, and RubyProf[57] doesn't disappoint. It can generate thread/method level reports that outline hotspots in our application.

57. http://ruby-prof.rubyforge.org

The trick though, is getting RubyProf to profile a controller action and then append the profile report to the action's response. So we need a convenient way to wrap an action with some profiling code. As if by design, an around_filter does just that:

```
around_filter do |controller, action|
  logger.debug "before #{controller.action_name}"
  action.call
  logger.debug "after #{controller.action_name}"
end
```

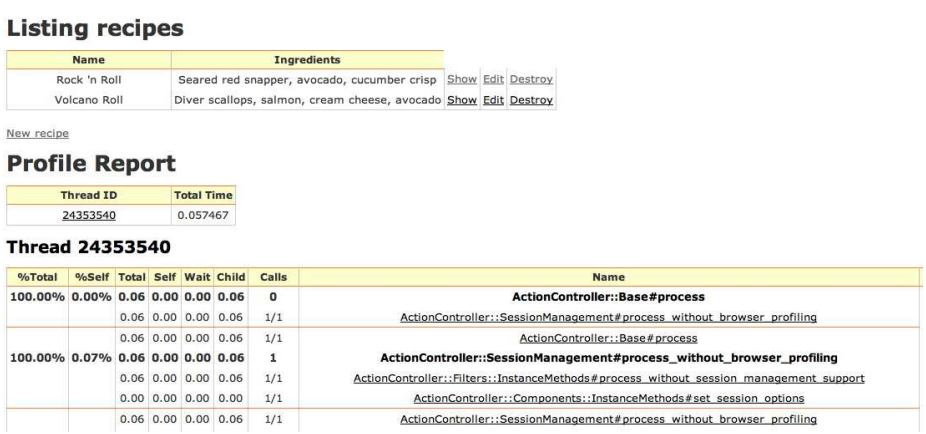
Mix these two ingredients together in our ApplicationController and we have ourselves an in-browser profiler for any request:

Download BrowserProfiling/app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  around_filter do |controller, action|
    if controller.params.key?("browser_profile!")
      require 'ruby-prof'

      # Profile only the action
      profile_results = RubyProf.profile { action.call }

      # Use RubyProf's built in HTML printer to format the results
      printer = RubyProf::GraphHtmlPrinter.new(profile_results)

      # Append the results to the HTML response
      controller.response.body << printer.print("", 0)
    else
      action.call
    end
  end
```

If any incoming request has a browser_profile! parameter, we still call the requested action but under the watchful eye of RubyProf. Then we tack the profile report on to the end of the response. For example, if we type in the following URL we see the inline report:

http://my-production-host.com/recipes?browser_profile!



Listing recipes

Name	Ingredients			
Rock 'n Roll	Seared red snapper, avocado, cucumber crisp	Show	Edit	Destroy
Volcano Roll	Diver scallops, salmon, cream cheese, avocado	Show	Edit	Destroy

New recipe

Profile Report

Thread ID	Total Time
24353540	0.057467

Thread 24353540

%Total	%Self	Total	Self	Wait	Child	Calls	Name
100.00%	0.00%	0.06	0.00	0.00	0.06	0	ActionController::Base#process
		0.06	0.00	0.00	0.06	1/1	ActionController::SessionManagement#process_without_browser_profiling
		0.06	0.00	0.00	0.06	1/1	ActionController::Base#process
100.00%	0.07%	0.06	0.00	0.00	0.06	1	ActionController::SessionManagement#process_without_browser_profiling
		0.06	0.00	0.00	0.06	1/1	ActionController::Filters::InstanceMethods#process_without_session_management_support
		0.00	0.00	0.00	0.00	1/1	ActionController::Components::InstanceMethods#set_session_options
		0.06	0.00	0.00	0.06	1/1	ActionController::SessionManagement#process_without_browser_profiling

There's just one small problem. Our around_filter is not guaranteed to be the first and last filter in the filter chain. That means some other before_filter or after_filter could be to blame for our performance problems, and we'd never know it. That is, until we rearrange our code slightly.

Instead of using an around_filter to wrap the action, we could wrap the entire request *process*. It involves basically the same code, this time added to ActionController::Base:

Download BrowserProfiling/lib/rails_extensions.rb

```
module ActionController
  class Base
    def process_with_browser_profiling(request, response,
                                       method = :perform_action,
                                       *arguments)

      if request.parameters.key?('browser_profile!')
        require 'ruby-prof'

        # Profile only the action
        profile_results = RubyProf.profile {
          response = process_without_browser_profiling(request, response,
                                                       method, *arguments)
        }

        # Use RubyProf's built in HTML printer to format the results
        printer = RubyProf::GraphHtmlPrinter.new(profile_results)

        # Append the results to the HTML response
        response.body << printer.print("", 0)

        # Reset the content length (for Rails 2.0)
        response.send("set_content_length!")
```

```
        response
      else
        process_without_browser_profiling(request, response,
                                           method, *arguments)
      end
    end
    alias_method_chain :process, :browser_profiling
  end
end
```

The key to making this work is the alias_method_chain method. When a request comes into our app, the process method is invoked to handle it. But we want to do that with profiling enabled. So alias_method_chain says that whenever the profile method is called, our profile_with_browser_profiling method should be called instead. Then, to invoke the original process method in our profiling block (or if the request shouldn't be profiled), we call process_without_browser_profiling.

Also See

While this recipe works for most scenarios, it doesn't support POST operations or redirects. To do that, we'd need to append the profiling results to a file instead of the HTML response. For a more full-featured profiler based on this recipe, check out the BrowserProfiler plugin[58]:

```
$ script/plugin install svn://rubyforge.org/var/svn/browser-prof
```

Also if you dig having inline reports, you might like the BrowserLogger plugin[59] which appends the current request's log to the end of a response. You'll never tail a log file again!

58. http://rubyforge.org/projects/browser-prof/
59. http://rubyforge.org/projects/browser-logger/

Caching Up With the Big Guys

By PJ Hyett and Chris Wanstrath (http://errtheblog.com)
By day, PJ Hyett and Chris Wanstrath run the Rails consulting and training firm, Err Free. By night, they maintain Err the Blog, a running commentary on both Ruby and Rails.

Problem

You've received massive funding. You've hit the front page of DiggCrunch. New users are flowing like champagne at your launch party. But despite using standard Rails template caching, your app is slowing, slowing, slowing down.

All that beautifully concise Active Record code spread throughout your app is now becoming a serious bottleneck as your tables (and bandwidth bills) start growing by the millions. Wouldn't it be great to change that clever code slightly into something simple and watch (most of) your problems disappear?

Ingredients

- The memcached daemon.[60] On a Mac we install it like this:

  ```
  $ sudo port install memcached
  ```

  There is also a port of memcached for the win32 architecture.[61]

- The memcache-client library:

  ```
  $ gem install memcache-client
  ```

- The cache_fu plugin:

  ```
  $ script/plugin install svn://errtheblog.com/svn/plugins/cache_fu
  ```

60. http://www.danga.com/memcached
61. http://jehiah.cz/projects/memcached-win32/

Solution

At some point, even the cleanest, most-efficient code can't handle massive loads—not because it's slow, but commonly because a resource it depends on is slow. For web apps, it's usually the database. That's where the gentle developers of LiveJournal come in. They wrote an awesome library by the name memcached of which they've this to say:[62]

"memcached is a high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load."

Sounds great, right? It may not cook your chicken rotisserie style in 10 minutes, but it has saved sites like Gamespot, Facebook, Wikipedia, and Digg from an early demise. And developers in the Rails community are working towards making memcached one of the easiest ways to scale your app. So let's get right to it, shall we?

First we start up the memcached daemon process:

```
$ rake memcached:start
```

Conceptually, memcached is just a distributed hash: it caches objects indexed by a key. The Rake task fires it up on a server and port combo specified in the config/memcached.yml file. Here's the default development configuration that was created when the cache_fu plugin was installed:

```
development:
  servers: localhost:11211
```

Cool, now let's get to the goods with something simple. Say we have an action that fetches a user along with all of their groovy tags:

```
class UsersController < ApplicationController
  def show
    @user = User.find(params[:id], :include => :tags)

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @user }
    end
  end
end
```

62. http://www.danga.com/memcached

It's very clean, but it's slowing down as our database and traffic grows. So let's cache each user in the memcached daemon's memory. To do that, we just add acts_as_cached to our User model:

```
class User < ActiveRecord::Base
  acts_as_cached :include => :tags
end
```

Then we replace the find method in the controller action with get_cache:

```
class UsersController < ApplicationController
  def show
    @user = User.get_cache(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @user }
    end
  end
end
```

That's all there is to it, mostly. Pretty simple object caching. The get_cache method will try to fetch data uniquely identified by the User model and value of params(:id) from the cache. If nothing is found (a *cache miss*), it will call User.find with the params(:id) value and cache the result. In other words, the first time the show action is hit a DB query is run. Here, let's have a look in the log:

```
User Load Including Associations (0.189375)  SELECT `users`.`id`...
==> Set User:694624473 to cache. (0.01550)
Completed in 4.63984 (0 reqs/sec) | Rendering: 0.00114 (0%) |
Memcache: 0.01745 | DB: 0.19800 (4%) | 200 OK
```

If something is found, it will just return it. That means when we hit the action a second time, there's no DB query:

```
==> Got User:694624473 from cache. (0.00983)
Rendering users/show
Completed in 0.02702 (37 reqs/sec) | Rendering: 0.00334 (12%) |
Memcache: 0.00983 | DB: 0.00281 (10%) | 200 OK
```

That's a good start! However, as our controllers become more skinny and our models fatter, we're going to end up with lots of custom finders. Say we have a page displaying the last 50 users that have signed up, and we want to show them along with their cute profile picture, and maybe their tags. Not a big deal, but this query involves three quite large tables. Here's our custom finder:

```
class User < ActiveRecord::Base
  def self.find_latest
    find :all, :order => 'users.id desc',
```

```
                :limit => 50,
                :include => [:picture, :tags]
  end
end
```

And here's our index action that uses it:

```
class UsersController < ApplicationController
  def index
    @users = User.find_latest

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @users }
    end
  end
end
```

Skinny controller, fat model? Check! Ordering by id? Check! Loading pictures and tags in bulk? Check! Query takes two seconds to execute? Check! Wait, that last check's no good.

So instead of running that two-second query on every page load, let's just run it once every five minutes and cache the result. To do that, we just change the acts_as_cached slightly in our User model. Here's our revised model in its entirety:

Download CachingUp/app/models/user.rb

```
class User < ActiveRecord::Base

  has_one :picture
  has_and_belongs_to_many :tags

  acts_as_cached :ttl => 5.minutes

  def self.find_latest
    find :all, :order => 'users.id desc',
               :limit => 50,
               :include => [:tags, :picture]
  end

end
```

The custom finder method in the model wasn't changed? That's right! The cache_fu plugin offers a number of really sweet features, and one of our favorites is being able to pass a method name to the cached method and have the result cached. For example, now over in our controller we can use User.cached(:find_latest):

Download CachingUp/app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  def index
    @users = User.cached(:find_latest)

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @users }
    end
  end
end
```

Calling cached(:find_latest) caches the result of the finder and puts it in the cache using the method name as the key.

This is the crux of caching. Databases are always the bottleneck in web applications, so you avoid that stress by putting as much of your data in memory as makes sense. The idea is that the data is retrieved from the database once, put into the cache, and returned from the cache on any subsequent requests until it has been invalidated.

Speaking of invalidation, how do we ensure our cache is fresh beyond setting an appropriate timeout? The way we have our User model set up now, the cache is expired every five minutes. Rails, conveniently, provides hooks into a model object's lifecycle, letting us automate the cache expiry process:

```
class User < ActiveRecord::Base
  acts_as_cached
  after_save :expire_cache
end
```

Now when a User object is updated, her record will be expired from the cache. When the next request comes in for that user, Rails will grab a fresh copy from the database and put the object right back in the cache where it belongs. This means we have strong cache integrity—we'll rarely get stale data.

However, if we think we have stale data and want to see our page generated straight from the database, we can write a before_filter to force a cache reset:

Download CachingUp/app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  before_filter :set_cache_override

private
```

```
  def set_cache_override
    returning true do
      ActsAsCached.skip_cache_gets =
        !!params[:skip_cache]
    end
  end

end
```

Setting skip_cache_gets tells cache_fu to treat every cache lookup as a cache miss. We can trigger it by hitting:

http://localhost:3000/users?skip_cache=1

But be very careful with this! If you do this on the front door of a big site during peak hours, all those expensive queries get re-run. A safer alternative is to call User.reset_cache, which grabs data and sets it in the cache without expiring the key. While this is going on, every request gets the old cached data.

Hey, this site is getting faster! But there's one more wrinkle. Let's say we also have a page that lists all the recent forum posts, and next to each post we show the user name and their picture. Some users appear more than once. So we need to load all the users who have posts on this page, but we don't want to cache users along with their forum posts. Otherwise every time a user changed her picture, we'd have to clear the cache of every post she made. Instead, we can pass a list of user ids into the get_cache method, like so:

```
user_ids = @posts.map(&:user_id).uniq
@users   = User.get_cache(user_ids)
```

This grabs all the matching keys in parallel, filling in the cache misses as it goes. When all is said and done, @users is a hash keyed by the user id and the value is the corresponding User model. Then we can iterate through the hash data in our view to keep things speedy.

Finally, it's time to put all this caching goodness into production. First we need to check the configuration of the production section of the config/memcached.yml file:

```
production:
  namespace: killer_app
  servers:
    - 192.185.254.121:11211
    - 192.185.254.138:11211
    - 192.185.254.160:11211
```

The namespace is important because it's the namespace all the keys live under, and setting it for our app lets us have different apps sharing the same memcached. It's also important to use IP addresses for where the memcached daemons live to avoid DNS requests.

Then once we have the production environment configured, we just fire up the memcached daemon on the production machines:

```
$ rake memcached:start
```

There's also a memcached_ctl script that comes with cache_fu for managing the daemon on multiple servers.

Using memcached isn't a silver bullet—there are a number of hardware and architectural considerations to keep in mind as your app grows. For example, if you can't keep up with the IO requests, then you won't benefit from memcached. And in smaller apps it may actually make things slower. Focus on your app first, then add cache_fu in later when you have evidence that you really need it.

Discussion

You can cache anything: generated images, intense number crunching, HTML, etc.

Also See

Using memcached adds at least one more moving part to your production environment. As reliable as memcached may be, it's wise to monitor the daemons across multiple servers using Monit. See Recipe 67, *Monitoring (and Repairing) Processes with Monit*, on page 285 for an example.

Dynamically Updating Cached Pages

By Mike Subelsky (http://www.subelsky.com/)
Mike Subelsky is a former Navy cryptologic officer and cybersecurity analyst now working as a freelance software developer and Rails hacker. He lives and hacks in Baltimore, Maryland.

Problem

You want to use page caching but still dynamically customize a few page elements for each user. For example, in the header of every cached page you want to show a "Login" link to users who haven't already logged in, and show the user's name and a "Logout" link if they are logged in.

Solution

Suppose at the top of every page in our application we make the logged in user feel warm and fuzzy, like this:

The dynamic part is contained within a login span in our application layout file:

Download DynamicCacheContent/app/views/layouts/application.html.erb

```
<span id="login">
  <% if logged_in? -%>
    Welcome, <%= h user_name %>!
    (<%= link_to 'Log Out', logout_path %>)
  <% else -%>
    <%= link_to 'Log In', login_path  %>
  <% end -%>
</span>
```

Now let's say the front page of our app gets a lot of action, so we enable page caching. The next request for the front page will cache it and subsequent requests will bypass Rails completely—the web server

merely serves the static page from our public directory. That adds up to lightning-fast page loads.

The only downside is that the static page is, well, static. So if Wally is already logged in and he's the first person to hit our front page, then everyone sees Wally's greeting. Worse yet, it appears as though everyone is logged in as Wally. That is unless we add a wee bit of JavaScript into the page. Then we can have our cake and eat it too!

As it stands, when a user logs in we put their user id in the session. That's how our *application* remembers that the user has already logged in. But we need the cached *page* to remember, too. To do that, when the user logs in we'll also create a cookie that contains the user's name. So in the controller that handles our login/logout functions, we'll have methods that look like this:

```
def login
  user = User.authenticate(params[:login], params[:password])
  if user
    reset_session
    session[:user_id]   = user.id
    cookies[:user_name] = {:value => user.name}
    redirect_to home_url
  else
    flash[:error] = "Sorry, try again."
    render :action => 'new'
  end
end

def logout
  reset_session
  cookies.delete :user_name
  redirect_to home_url
end
```

Next we need some JavaScript that looks for a named cookie and dynamically updates the personalized area. As we're already using the Prototype library for other features, we'll use it here for simplicity:

Download DynamicCacheContent/public/javascripts/application.js

```
var LoginCheck = Class.create({
  initialize: function(cookie_name) {
    var cookie_value = get_cookie(cookie_name);
    if (!cookie_value) {
      $('login').update("<a href=\"/login\">Log In</a>");
    } else {
      $('login').update("Welcome, " + cookie_value.escapeHTML() + "!" +
                        " (<a href=\"/logout\">Log Out</a>)");
    }
```

```
  }
});

function get_cookie(name) {
  var value = null;
  document.cookie.split('; ').each(function(cookie) {
    var name_value = cookie.split('=');
    if (name_value[0] == name) {
      value = name_value[1];
    }
  });
  return value;
}
```

If the user_name cookie is found in the browser, then we update the login element in our header to contain a "Log Out" link with the user's name. If no user_name cookie is found, we update the page to show the "Log In" link instead.

Finally, we just call the JavaScript function at the bottom of our application layout file, passing in the name of the cookie we're looking for:

Download DynamicCacheContent/app/views/layouts/application.html.erb

```
<script type="text/javascript">
  // <![CDATA[
  new LoginCheck('user_name');
  // ]]>
</script>
```

Now when the cached page is written to disk, the page includes the JavaScript which dynamically updates the links on the page.

To test this in development mode, remember to (temporarily) enable caching in your config/environments/development.rb file:

```
config.action_controller.perform_caching = true
```

Discussion

You could also use Prototype AJAX calls back to your server to perform more complex, dynamic page updates.

You'll likely want to name your cookie something unique, rather than just user_name.

Also See

- If you need to cache pages that contain flash messages, check out the Cacheable Flash plugin.[63]

63. http://www.pivotalblabs.com/articles/2007/08/08/cacheable-flash

Part XI

Security Recipes

Flipping On SSL

Problem

You need a declarative way of specifying that certain actions should only be allowed to run under SSL. If they're accessed without it, they should be redirected.

Ingredients

- David Heinemeier Hansson's ssl_requirement plugin:

```
$ script/plugin install ssl_requirement
```

Solution

The solution is easier than you might imagine, but involves two deft steps: configuring our web server and marking up our controllers. Let's get the web server out of the way first, and save the dessert for last.

We've got our trusty web server listening on port 443, and along comes an SSL request. If it's a dynamic request, the web server pawns if off to our Rails app. Rails, however, needs to know if the original request came in via https. It's the web server's job to send along this piece of information to our Rails app, and the HTTP way to do that is by setting a header.

As it turns out, Rails is already waiting for an HTTP header called X_FORWARDED_PROTO. So in our Apache httpd.conf file, we just set the X_FORWARDED_PROTO header to https in the virtual host for port 443. You'll have more stuff in your file, but here are the relevant parts we need to set:

```
<VirtualHost *:443>
  SSLEngine on
  RequestHeader set X_FORWARDED_PROTO "https"
</VirtualHost>
```

If you're using the nginx web server, you'll set the header slightly different:

```
server {
  listen 443;
  location / {
```

```
    proxy_set_header X_FORWARDED_PROTO https;
  }
}
```

OK, that's part one. Our Rails app now knows when an SSL request comes in, and we can check for it using the request.ssl? method. Part two is to do something with that information. Specifically, we want to be able to declare that certain actions must be run under SSL and have Rails "flip" to https when appropriate. That's where the plugin comes in. We just need to include it in our ApplicationController:

Download buffet/app/controllers/application.rb

```
include SslRequirement
```

This adds two methods to all our controllers: ssl_required and ssl_allowed. It also adds a before_filter that checks request.ssl?. If SSL is required for an action, but request.ssl? is false, then we'll get a redirect to the same URL but with the https protocol. The reverse is also true—running an action under SSL that doesn't require it will redirect to the http protocol.

Now, we don't want any of this to happen when we're issuing requests against our local development (or test) app. So while we're in the ApplicationController file, let's add this method to include the local machine check and then run the default checks:

Download buffet/app/controllers/application.rb

```
def ssl_required?
  return false if local_request? || RAILS_ENV == 'test'
  super
end
```

At this point we've added all the plumbing, so it's time for a reward. Over in our controllers, here's what we can do now:

Download buffet/app/controllers/accounts_controller.rb

```
class AccountsController < ApplicationController

  ssl_required :create, :change_password
  ssl_allowed :show

  def create
    # Non-SSL access will be redirected to SSL
  end

  def change_password
    # Non-SSL access will be redirected to SSL
  end
```

```
  def show
    # Works either with or without SSL
  end

  def index
    # SSL access will be redirected to non-SSL
  end

end
```

Creating accounts and changing passwords involves passing sensitive information around, so we ensure SSL is required for those actions. If the incoming request for the those actions isn't using the https protocol, then the request will be redirected to https. Showing an account doesn't display any sensitive information, but SSL *can* be used. In other words, it won't redirect to http if the incoming request is https. And, in this case, the listing should always be on an http connection, so we don't have to declare anything for it.

Before we go, there's one subtle, but important, thing we need to check. By default, Rails will look for static assets (images, JavaScript files, etc.) in the public directory. But say we've configured our production environment to link these assets from a dedicated server, like so:

```
config.action_controller.asset_host = "http://my-assets.com"
```

Since we've hard-coded the protocol, actions run under SSL will have assets linked via http. Some browsers pop up a security warning when this happens, and consequently users get a little freaked. So to make sure all the assets are automatically linked using the same protocol as the incoming request, we need to remove the explicit protocol:

```
config.action_controller.asset_host = "my-assets.com"
```

Discussion

The SslRequirement module adds the before_filter at the point at which it's included. If you want to run other filters before that, you must declare them ahead of including the module.

There are other valid reasons for overwriting the ssl_required? method. For example, you could inspect an @account variable and always flip to SSL if it's a premium account. That is, you don't have to rely solely on the declarative style.

Locking Down Sensitive Data

If you can read only one recipe in this book, make it this one! The steps are trivial, but oh! so easy to forget. And the cost of forgetting—well, you don't want to find out.

Here's an innocent looking model:

```
class User < ActiveRecord::Base
end
```

The users table has a number of boolean columns, including is_admin and gets_free_orders. Of course we don't put those on the account creation form, otherwise anyone can make themselves special. But as the model stands, they don't need no stinkin' form to rule our system!

Anyone with a 'net connection and a frisky spirit can just post values for those columns to our create action:

```
@user = User.new(params[:user])
```

It's an easy fix though. We can protect sensitive attributes from bulk assignment by using attr_accessible in our model to only name the attributes that are available for bulk-assignment:[64]

Download events/app/models/user.rb

```
attr_accessible :name, :email, :password
```

Now to upgrade a user, we'd have to make an explicit assignment in code, like so:

```
@user.is_admin = true
@user.save
```

But don't put this recipe down just yet!

Here's an innocent looking action:

```
def index
  @users = User.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @users }
```

64. The attr_protected method does the reverse, and in so doing leaves the door open if you add new columns. It's generally better to use positive access control with attr_accessible.

```
  end
end
```

Of course we don't show any sensitive user information on the HTML page, even if it's viewable by admins only. But as the action stands, prying eyes don't need no stinkin' HTML page to compromise our data! They'll just send in the xml format and read the data in raw XML form.

It's an easy fix though. We just override the to_xml method in our model to only spit out certain attributes:

Download events/app/models/user.rb

```
def to_xml(options = {})
  default_only = [:id, :name, :email]
  options[:only] = (options[:only] || []) + default_only
  super(options)
end
```

Do yourself a big favor. Spend a couple minutes having a look through your models and your tables (yes, *especially* if your app is already in production). Add attr_accessible and define to_xml where appropriate. You don't have to tell anybody you did....

Part XII

Deployment and Capistrano Recipes

Custom Maintenance Pages

Problem

Bad things sometimes happen to good applications. When you need to put out a fire (or do maintenance chores), you want to quickly put up a maintenance page then get right to work. And you want the temporary page to include your familiar logo, award-winning web design, and a little message that shows you care. Then when you've got everything under control, you want to put the application back online just as quickly as you took it down.

Ingredients

- The Capistrano[65] gem (version 2.1.0+):

```
$ gem install capistrano
```

Solution

If you've had the pleasure of using Capistrano[66] then you know it lives to serve you. Need to put up a maintenance page in a hurry? Capistrano's got your back. Rush over to your keyboard and type:

```
$ cap deploy:web:disable
```

And when the klaxons stop blaring and the birds start chirping again, taking down the maintenance page is equally satisfying:

```
$ cap deploy:web:enable
```

Great, now for the customization. The standard maintenance page that comes with Capistrano works in a pinch, but it's easy to create a custom maintenance page that sets our app apart from the crowd. Now we're no famous web designer, but the guy down the hall is. And all we need is a Rails template file that shows our logo, some excuse for the site being down, and an indication when it might return. So he whips up this one:

65. http://www.capify.org/
66. http://www.capify.org/

Download capistrano/app/views/admin/maintenance.html.erb

```
<html xmlns="http://www.w3.org/1999/xhtml"
      version="-//W3C//DTD XHTML 1.1//EN" xml:lang="en">

<head>
  <title>Custom Maintenance Page</title>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <link href="/stylesheets/maintenance.css" rel="stylesheet" type="text/css" />
</head>

<body>
  <div id="content">
    <img src="http://demo.pragmaticstudio.com/images/fr_arr.jpg"
         alt="Recipes" />
    <h1>
      We're currently offline for <%= reason ? reason : "maintenance" %>
      as of <%= Time.now.strftime("%I:%M %p %Z") %>.
    </h1>
    <p>
      Sorry for the inconvenience.  We'll be back
      <%= deadline ? "by #{deadline}" : "shortly" %>.
      Please <a href="mailto:info@railsrecipes.com">e-mail us</a>
      if you need to get in touch.
    </p>
  </div>
</body>
</html>
```

There's nothing extraordinary about this template, but notice the two variables reason and deadline. That's our dynamic content: why the site is down and when it'll return. So clearly we need to run our hand-crafted template through the ERb templating system to get a static HTML file. Then we need to upload it to our web servers. It turns out that's exactly what Capistrano does with its stock template when we run the deploy:web:disable task. We just need to override the default task in our config/deploy.rb recipe file:

Download capistrano/config/deploy.rb

```
namespace :deploy do
  namespace :web do

    desc "Serve up a custom maintenance page."
    task :disable, :roles => :web do
      require 'erb'
      on_rollback { run "rm #{shared_path}/system/maintenance.html" }

      reason      = ENV['REASON']
      deadline    = ENV['UNTIL']
```

```
      template = File.read("app/views/admin/maintenance.html.erb")
      page = ERB.new(template).result(binding)

      put page, "#{shared_path}/system/maintenance.html",
                :mode => 0644
    end
  end

end
```

Did you know you could use ERb directly like that? It's easy: Just read in a template file and let ERb render it into a variable. There's a neat trick here, too. The local variables reason and deadline are set based on environment variables. To make them accessible to our template, we hand ERb the current binding.

Now we have the maintenance page template all filled out and in memory. To upload it into a maintenance.html file on all the production web servers, we use Capistrano's put command.

Even if we were to run the deploy:web:disable task right now, it wouldn't disable access to our application. We'd end up with a system/maintenance.html file in our Rails app's public directly, but you'd only see if it you typed the file name into the browser. Instead, we need it to be shown whenever *any* dynamic request is made to our application.

The Apache web server has an extremely powerful URL rewriting engine called mod_rewrite[67]. This works perfectly for what we're trying to do and takes just four lines of web server configuration:

```
RewriteCond %{DOCUMENT_ROOT}/system/maintenance.html -f
RewriteCond %{REQUEST_URI} !\.(css|jpg|gif|png)$
RewriteCond %{SCRIPT_FILENAME} !maintenance.html
RewriteRule ^.*$ %{DOCUMENT_ROOT}/system/maintenance.html [L]
```

Basically this says to redirect all incoming requests (except those for our external CSS file and logo) to the static system/maintenance.html file if it exists.

We can do the same thing with the Nginx web server:

```
if ($request_filename ~* \.(css|jpg|gif|png)$) {
  break;
}
if (-f $document_root/system/maintenance.html) {
  rewrite  ^(.*)$  /system/maintenance.html last;
  break;
```

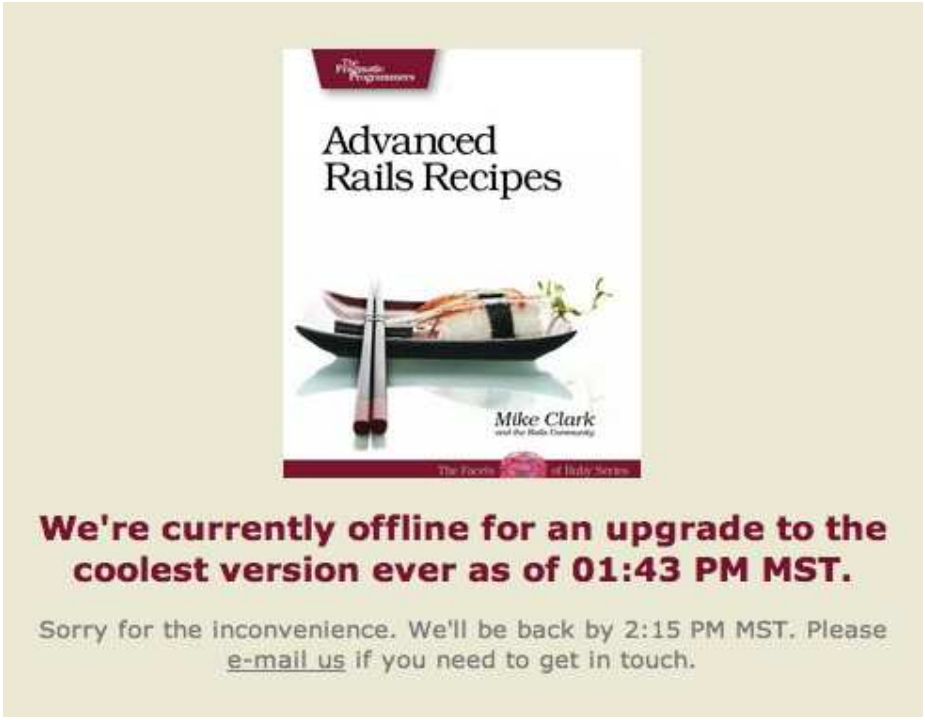
67. http://httpd.apache.org/docs/1.3/mod/mod_rewrite.html

```
}
```

Now let's say we've been building up to a brand new version of our app, and it's so much better that we need to take the application offline to do a bunch of system stuff. It'll take about 30 minutes. Here's where those environment variables (and the guy down the hall) pay off:

```
$ REASON="an upgrade to the coolest version ever" \
  UNTIL="10:30 AM MST" \
  cap deploy:web:disable
```

Hitting any Rails action shows our custom maintenance page:



Discussion

You might be inclined to try using Rails helper methods, such as time_ago_in_words, in your maintenance page template. Sorry, this won't work because Capistrano doesn't load the Rails framework when you call tasks. Then again, if you need helpers in your maintenance page, perhaps it's doing too much.

Running Multi-Stage Deployments

Problem

You're using Capistrano to deploy your application into production, and now you need to deploy the same application to different environments: staging, testing, bobs_mac, favorite_clients_box, etc.

Ingredients

- The capistrano gem (version 2.1.0+):

```
$ gem install capistrano
```

- The capistrano-ext gem:

```
$ gem install capistrano-ext
```

Solution

The solution is fairly trivial, but gives insight into the flexibility of Capistrano recipe files and how they run. When we're done you may well find other ways to improve your deployment recipes.

Normally when we're just deploying into one environment, we define a single stanza of roles like this:

Download capistrano/config/deploy.rb

```
role :web, 'railsrecipes.com'
role :app, 'railsrecipes.com'
role :db,  'railsrecipes.com', :primary => true
```

But that won't work if we have multiple environments because the roles will always be set to our production machines. Instead, we need to define the roles on a per-environment basis. An easy way to do that is using a task definition. So for two roles—staging and production—we need to add two tasks to our recipe file:

Download capistrano/config/deploy.rb

```
task :staging do
  role :web, 'staging.railsrecipes.com'
  role :app, 'staging.railsrecipes.com'
```

```
  role :db,  'staging.railsrecipes.com', :primary => true
  set  :stage, :staging
end

task :production do
  role :web, 'railsrecipes.com'
  role :app, 'railsrecipes.com'
  role :db,  'railsrecipes.com', :primary => true
  set  :stage, :production
end
```

Then to set up the roles for a specific environment, we call the corresponding environment task before the task that does the real work:

```
$ cap staging deploy
```

```
$ cap production deploy
```

Now let's say we want to change the deployment directory based on the environment. That is, we need to change the following line to include the environment name in the path:

```
set :deploy_to, "/path/to/#{application}"
```

We might be tempted just to use the stage variable in the string, but there's a subtle reason this won't work. The deploy_to variable is evaluated when our recipe is loaded, and we need to defer the evaluation until after we've set the environment up. Typing a couple extra characters to create a proc is all it takes to cause the variable be evaluated lazily:

```
set(:deploy_to) { "/path/to/#{application}/#{stage}" }
```

As a do-it-yourself approach, all this works great. In fact this idiom became so common that Jamis Buck packaged it up in the capistrano-ext gem. Now that you know how it works, let's give it a whirl. With the gem installed, we just need to add this to the top of our recipe file:

Download capistrano/config/deploy.rb

```
set :stages, %w(staging testing production bobs_mac)
set :default_stage, 'staging'
require 'capistrano/ext/multistage'
```

Now we can clean up some code. We put code that's specific to the testing stage, for example, in the config/deploy/testing.rb file:

Download capistrano/config/deploy/testing.rb

```
role :web, 'testing.railsrecipes.com'
role :app, 'testing.railsrecipes.com'
role :db,  'testing.railsrecipes.com', :primary => true
```

```
set :deploy_to, "/path/to/#{application}/testing"
```

Notice here we don't need to lazily evaluate the deploy_to variable because of the order in which the files are loaded. As long as the :application variable is set *before* the capistrano/ext/multistage recipe is loaded. (In general, it's probably good practice to put all such require statements at the end of your deploy.rb file anyway.)

To deploy to the testing environment, we run

```
$ cap testing deploy
```

And if we don't specify an environment, we'll deploy to the staging environment by default. If you don't set :default_stage, you'll get an error if you try to do anything without explicitly specifying a stage. Some people prefer this to having a default stage. If you do set a default stage, it's always a good ideas to pick a default other than production. That way you won't accidentally push new code out into the wild, wooly web. I knew a guy who did that once...

Discussion

Creating New Environments

Problem

You need to run your application in various modes which have environment-specific settings, and the default Rails environments aren't enough.

Solution

There's nothing special about the default runtime environments—development, test, and production—except that most projects need those at a minimum. That doesn't stop us from adding new environments. If we have special requirements for our Rails app when it's running on the staging servers, for example, we just add a staging.rb file to the config/environments directory:

Download buffet/config/environments/staging.rb

```
config.cache_classes = true
config.action_controller.consider_all_requests_local = false
config.action_controller.perform_caching             = true

GATEWAY_URL='https://test.authorize.net/gateway/transact.dll'
```

For the most part it's the same as the production-level configuration, but there's a twist. In production this particular application charges credit cards via an external payment gateway. In staging we don't want to be charging *real* credit cards, so we set the GATEWAY_URL variable to point to a test server. In production.rb, it points to the honest-to-goodness live server.

Before going any further, we also need to configure a database for the staging environment to use. That's as easy as adding a staging stanza to our config/database.yml file:

Download buffet/config/database.yml

```
staging:
  adapter: mysql
  encoding: utf8
  database: buffet_staging
  username: stage
  password: fright
```

Now when we run with staging as the current environment, Rails will load staging.rb instead of one of the default environment files and use the staging database. Let's try it first in the console:

```
$ ruby script/console staging
Loading staging environment
>> RAILS_ENV
=> "staging"
>> GATEWAY_URL
=> "https://test.authorize.net/gateway/transact.dll"
```

That's a good sanity check that our staging.rb is getting picked up. Now let's fire up the app with its staging face on:

```
$ ruby script/server -e staging
** Starting Rails with staging environment...
```

The way we set the staging environment when running the app depends on how we start it. If we're using the mongrel_cluster gem to run a pack of Rails apps, for example, we'd need to set the environment in the cluster configuration file. You get the idea. Wherever we set production before will change to staging.

Remember that some of the built-in Rake tasks rely on the RAILS_ENV environment variable to know which database to use. (The default is development.) So if we're running migrations in the staging environment, we need to call it out:

```
$ RAILS_ENV=staging rake db:migrate
```

We can simplify that a bit by writing a custom Rake task that sets RAILS_ENV for us and invokes the :environment task to load Rails in proper environment:

Download buffet/lib/tasks/environments.rake

```
desc "Sets the environment variable RAILS_ENV='staging'."
task :staging do
  ENV['RAILS_ENV'] = RAILS_ENV = 'staging'
  Rake::Task[:environment].invoke
end

desc "Sets the environment variable RAILS_ENV='production'."
task :production do
  ENV['RAILS_ENV'] = RAILS_ENV = 'production'
  Rake::Task[:environment].invoke
end
```

This spares us a few keystrokes every time we run a Rake task that needs the environment name:

```
$ rake staging db:migrate
```

Discussion

You can take this as far as necessary with multiple environments and special configuration values for each.

Also See

- What if you want to deploy to different machines depending on the Rails environment? See Recipe 52, *Running Multi-Stage Deployments*, on page 237 to learn how to set up Capistrano for multi-stage environments.

Managing Plugin Versions

Problem

You built your application with the help of some Rails plugins. Going forward you want to keep those plugins up to date, but on your own terms. And the last time you used svn:externals to link in a plugin, the remote Subversion repository you linked to went for a long vacation just as you were trying to deploy your app. So you need to take control of the plugin code and still be able to update to new revisions when you're ready.

Ingredients

- The Piston gem:

```
$ gem install piston
```

Solution

Piston[68] lets us import plugins into our local Subversion repository and sync them up with their master copy whenever we want. Yes, Piston is awesome. So let's pistonize a plugin already.

Say our app needs some super-duper pagination—I always reach for the will_paginate plugin. First we import it straight away into our Subversion repository:

```
$ cd vendor/plugins
$ piston import svn://errtheblog.com/svn/plugins/will_paginate
```

That exports the plugin into the vendor/plugins/will_paginate directory. To finish the import, we just need to commit the new files:

```
$ svn commit -m 'Importing local copy' vendor/plugins/will_paginate
```

What we have now is our own private copy of the plugin in our local repository. We can even modify the plugin code, check in our changes, and manage the plugin just like our application code. So far so good.

Now normally taking a copy of code like this means we couldn't easily sync up with future revisions of the plugin. But Piston keeps a little

68. http://piston.rubyforge.org/

secret for us: it remembers where the code came from. Imagine that one day the good folks who created the will_paginate plugin release a tasty new version. It happens to be a slow day around the office and we're itchin' to try something new, so we get the latest revision:

```
$ piston update vendor/plugins/will_paginate
Processing 'vendor/plugins/will_paginate'...
  Fetching remote repository's latest revision and UUID
  ...
  Updated to r413 (2 changes)
```

This merges any of our local changes with the latest revision that's on the remote repository. (If for reason there's a conflict, Piston doesn't detect it, but Subversion will reject the next commit.) In other words, our changes are preserved when we take new updates. After running the tests, we check the updates back into our local repository:

```
$ svn commit -m 'Updated to latest version' vendor/plugins/will_paginate
```

Then we remember that a new version of our application is getting rolled out at the end of the week. It would be a serious bummer if another new version of the will_paginate plugin was released and the new guy on our project mistakenly merged it into our version right before the deployment. So let's prevent that from happening by locking the version we tested:

```
$ piston lock vendor/plugins/will_paginate
```

Pistonizing plugins is so easy that before the deployment we go ahead and pistonize all the plugins our app depends on. That way it won't matter if Bob's Basement Plugins(TM) repository decides to curl up and die while we're trying to deploy.

Then later on when we want to sync back up with the latest plugin revision on its remote repository, we can unlock our copy and update it:

```
$ piston unlock vendor/plugins/will_paginate
$ piston update vendor/plugins/will_paginate
$ svn commit -m 'Updated again' vendor/plugins/will_paginate
```

And we can run piston update in our top-level plugins directory if we want to update all piston-managed plugins.

Piston makes light work of plugin dependency management. Just type a couple of commands and you're on to other tasks. And once you've pistonized a plugin, folks on your project who don't have the Piston

gem installed can happily check out and update the plugin via your local Subversion repository.

Discussion

If you're currently using svn:externals to manage your plugins, you can use piston convert to convert them to Piston-managed folders.

One important caveat: Piston doesn't preserve change history from the remote repository. Piston just takes the latest revision, or differences between what you currently have and the latest revision, and merge those changes into your checked out copy. However, you can examine the changes before committing them to your local repository.

Also See

Clearly you could also use Piston to manage the version of Rails in your vendor/rails directory. However, the update process that Piston uses is known to be slow. This usually isn't a big deal with plugins because they tend to be relatively small chunks of code that don't change all that often. However, updating and storing local copies of Rails in each application you write feels a tad heavy-handed. I usually just keep versions of Rails checked out on the production servers, and symlink the application's vendor/rails directory to a specific version after deployment using a Capistrano after hook.

Safeguarding the Launch Codes

You really don't want *the* production database password rattling around on your development machine. You know, the laptop you take with you to software conferences and the local Hackers Anonymous meetings. But how do you deploy without the launch codes?

Well, let's start by putting the real database.yml file, the one with all the secrets, in one place: our production box. Then we can restrict access to it using accounts, permissions, and all that good operating system stuff. This file will not be checked in to our regular project—the project will use a version that simply gives access to the development and testing databases.

Then when it comes time to push the deploy button, we'll let Capistrano[69] copy the real database.yml file into place for us. All we need is a hook that automatically gets triggered after the latest version of our code has been checked out to the release_path directory. We'll add it to the deploy namespace of our deploy.rb file, like so:

Download capistrano/config/deploy.rb

```
namespace :deploy do

  task :copy_database_configuration do
    production_db_config = "/path/to/production.database.yml"
    run "cp #{production_db_config} #{release_path}/config/database.yml"
  end

  after "deploy:update_code", "deploy:copy_database_configuration"
end
```

Remember that run executes the given command on the *remote* machine. So our database.yml file gets slipped into place just before the application is restarted, and our hands are clean.

69. http://www.capify.org/

Config Files On-The-Fly

Jamie Orchard-Hays has developed Web applications since the late 1990s. He's worked in ASP, ColdFusion, JSPs, Tapestry, and Ruby on Rails. He created the Elemental plugin for Rails and contributed the Hpricot Mapper to Solr-Ruby. Currently he resides in beautiful Charlottesville, Virginia.

You need to set up your deployment servers with various and sundry configuration files: Mongrel, Apache, nginx, and so on. Since these files often share bits of information, such as the name of your app, keeping them in sync is tedious and prone to error. Wouldn't it be nice to make this all a part of a repeatable deployment step?

Capistrano[70] to the rescue (again)! Let's take the case where we want to create a Mongrel cluster configuration file and drop it into our remote server directory. We don't need a local file to do that. Instead, we can generate the configuration bits inside our deploy.rb recipe and add it to a task in the deploy namespace, like so:

Download capistrano/config/deploy.rb

```
namespace :deploy do

  task :upload_cluster_configuration, :roles => :app do
    cluster_config = <<-CMD
      port: 8000
      servers: 4
      address: 127.0.0.1
      cwd: #{deploy_to}/current
      pid_file: tmp/pids/#{application}-mongrel.pid
      user: capistrano
      group: capistrano
      environment: production
    CMD
    put cluster_config, "#{release_path}/config/mongrel_cluster.yml"
  end

  after "deploy:update_code", "deploy:upload_cluster_configuration"
end
```

One benefit of assembling the configuration on the fly like this is being able to reference existing variables such as application and deploy_to in our recipe to keep things dry.

70. http://www.capify.org/

The secret ingredient in this recipe is the put command. It effectively uploads the data (our configuration info) to the given file location. Then last, but by no means least, this is hooked so it runs after the update_code task.

So now, every time we run cap deploy, we *know* our Mongrels will restart with a consistent configuration. Now, imagine how easy setting up a new deployment box might be if we had all our configuration files generated this way.

Preserving Files Between Deployments

One of the obvious joys of using Capistrano[71] is the deploy task. It puts the current version of your app in a brand new directory on the server, then restarts everything from there. But if your app stores user-uploaded pictures, search engine indexes, and other artifacts in the current deployment directory, you lose them when you redeploy.

The solution? Put files you need to preserve across deployments in the shared directory rather than RAILS_ROOT. Then during the deployment process, have Capistrano link the shared assets into your current deployment directory. Here are some example tasks:

Download capistrano/config/deploy.rb

```
namespace :assets do

  task :symlink, :roles => :app do
    assets.create_dirs
    run <<-CMD
      rm -rf  #{release_path}/index &&
      rm -rf  #{release_path}/public/images/pictures &&
      ln -nfs #{shared_path}/index #{release_path}/index &&
      ln -nfs #{shared_path}/pictures #{release_path}/public/images/pictures
    CMD
  end

  task :create_dirs, :roles => :app do
    %w(index pictures).each do |name|
      run "mkdir -p #{shared_path}/#{name}"
    end
  end
end

after "deploy:update_code", "assets:symlink"
```

Notice that we can chains tasks together: the symlink task first invokes the create_dirs task to make sure the shared directories exist. Then we use symbolic links to point our current release to the shared search

71. http://www.capify.org/

indexes and pictures. The after hook makes sure every deployment has all the previously created goodies.

Responding To Remote Prompts

Thanks to Jamis Buck for the idea and technical bits for this recipe.

Sometimes, the remote machines you're controlling with Capistrano talk back, and even go so far as to ask a question! When that happens, how do you answer via your local terminal?

The answer lies in a special use of the venerable run method. You generally use it to fire off a command to all the servers in a given role. However, you can also hang a block off the run call and it'll get invoked when the remote process responds. You get three block parameters:

- channel is the SSH channel on which you can send data back to the remote process
- stream identifies the response stream as :err or :out
- output is, you guessed it, the data that was output from the remote process

The run method gets us close to a solution, but when a question comes in we need to prompt for the answer in our local console. That requires one more ingredient: Capistrano uses the HighLine[72] library to process local console input and output. We can use it, too, simply by accessing the underlying ui object.

Knowing all that, we can write a generic method that takes the name of the command we want to run and the question we expect it to ask us:

```
def run_with_prompt(command, expected_question)
  run command, :once => true do |channel, stream, output|
    if output =~ /#{expected_question}/
      answer = Capistrano::CLI.ui.ask(expected_question)
      channel.send_data(answer + "\n")
    else
      # allow the default callback to be processed
      Capistrano::Configuration.default_io_proc.call(channel, stream, output)
    end
  end
end
```

72. http://rubyforge.org/projects/highline/

Note here that the ui.ask method starts the interactive prompt if we get the question we expect. Also, we used the :once => true option on the run method so that it only runs on a single remote host. You'll want to cache the response and reuse it for similar prompts if you're running in a multi-machine environment.

Then we can reuse the method inside any task. For example, if we're updating packages using apt-get, we can respond to its specific question:

```
task :app_get_update, :roles => :app do
  run_with_prompt("apt-get update", "Are you sure?")
end
```

As one more interesting use of the run method with a stream (thanks to Chris Wanstrath[73]), this task starts up script/console on a remote server:

Download capistrano/config/deploy.rb

```
desc "Open script/console on the remote machine"
task :console, :roles => :app do
  input = ''
  cmd = "cd #{current_path} && ./script/console #{ENV['RAILS_ENV']}"
  run cmd, :once => true do |channel, stream, data|
    next if data.chomp == input.chomp || data.chomp == ''
    print data
    channel.send_data(input = $stdin.gets) if data =~ /^(>|\?)>/
  end
end
```

73. http://errtheblog.com/posts/19-streaming-capistrano

Generating Custom Error Pages

By Giles Bowkett (http://gilesbowkett.blogspot.com)
Giles Bowkett is a programmer, actor, screenwriter, DJ, musician, artist, blogger, and dharma bum. Originally from Chicago, he now lives in Hollywood, Silicon Valley, the northern New Mexico wilderness, and Black Rock City. John Dewey is a Ruby developer for the Los Angeles Times, and better at tetherball than you (unless there's ropies).

Problem

You want a consistent design across your error pages, and you want to be able to maintain this design easily in the face of change. The standard Rails approach to template reuse—partials and layouts—is great, but you can't use them in the dynamic way because you can't write error pages for Rails which depend on Rails working.

Ingredients

- The Capistrano gem:[74]

```
$ gem install capistrano
```

Solution

In Recipe 51, *Custom Maintenance Pages*, on page 233 we learned how to use Capistrano and ERb to deploy custom maintenance pages. Using the same combination of power tools, we can cook up a quick and light implementation of partials and layouts for our error pages.

Start with a simple CSS file. You'll want to base it on the general CSS for your application, but use only a subset necessary for the design of your error pages. You want your CSS cleanly separated from the CSS within Rails. Even if you're not using dynamic options like Sass[75] or CSS in ERb, in some error states your system will redirect every request to the error page, including requests for CSS files.

74. http://www.capify.org/
75. http://haml.hamptoncatlin.com/docs/sass

Next comes the layout. We can use ERb syntax, but without Rails' helper methods, so things like link_to and the JavaScript helpers are off limits. Just create a skeleton HTML document and add in standard ERb-quoted code. Here's an example layout:

Download capistrano/config/deploy/errors/error.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">

<html lang="en">
  <head>
    <title>
      <%= title %>
    </title>
    <style type="text/css">
      <%= File.read(stylesheet) %>
    </style>
  </head>
  <body>
    <div>
      <h1>
        <%= heading %>
      </h1>
    </div>
    <div>
      <%= body %>
    </div>
  </body>
</html>
```

We'll put this error.html.erb file, and our errors.css stylesheet file, in the config/deploy/errors directory.

Then we need to create a partial for each error condition. These will just include the error-specific HTML we want to include in the page. Here are a few examples:

Download capistrano/config/deploy/errors/_404.html.erb

```
<p>
  We're terribly sorry, but we couldn't find that page.
</p>
```

Download capistrano/config/deploy/errors/_500.html.erb

```
<p>
  Something failed spectacularly.  Of course, it's all our fault!
  The klaxons are blaring and we'll fix it promptly.
</p>
```

These files go in the config/deploy/errors directory, as well.

OK, next we need to render these error templates. We don't have Rails to do it for us, so we'll just create a simple ERb template engine in a method of our Capistrano config/deploy.rb recipe file:

Download capistrano/config/deploy.rb

```
def template_engine(template, partial=nil, stylesheet=nil, opts={})
  require 'erb'
  unless opts.empty?
    set :title, opts[:title]
    set :heading, opts[:heading]
    set :body, ERB.new(File.read(partial)).result(binding)
  end
  ERB.new(File.read(template)).result(binding)
end
```

This code first takes a hash and turns it into variables, then passes the partial and its template to ERb to combine them all together. When it passes ERb a binding, it gives ERb access to the variables we define here. Because the current binding in this code will contain all these variables being set, we can use the same variables in both our layouts and our partials. This means that when we set the title variable, for example, we can put it in the HTML title attribute of our error page and in a nice readable header, as well.

Once the error pages have been generated, we want to upload them to our production servers. We can automate the entire process with a Capistrano task:

Download capistrano/config/deploy.rb

```
def error_template_path(filename)
  ["config", "deploy", "errors", filename].join("/")
end

desc "Create error pages."
task :create_error_pages, :roles => [ :web, :app ] do
  errors = {
    "404" => { "title" => "Page Not Found",
               "heading" => "Page Not Found" },
    "422" => { "title" => "Oops!",
               "heading" => "The data you submitted was invalid." },
    "500" => { "title" => "Oops!",
               "heading" => "Kaboom!" }
  }
  errors.each_key do |error|
    template   = error_template_path("error.html.erb")
    partial    = error_template_path("_#{error}.html.erb")
    stylesheet = error_template_path("error.css")
    put template_engine(template, partial, stylesheet,
```

```
                :title => errors[error]["title"],
                :heading => errors[error]["heading"]
              ), "#{current_path}/public/#{error}.html",
              :mode => 0644
  end
end
```

In the create_error_pages task—which runs for both the :web and :app roles—we first specify a nested hash of error codes, titles, and headings.[76] We send that to the template engine, along with the filenames expanded into error template paths by the error_template_path method. Finally, the put command uploads the error page contents into their respective files in the public directory on the server.

Now we have a convenient way to turn ERb templates into real files on the server. For example, we could use the template_engine method to deploy maintenance pages as described in Recipe 51, *Custom Maintenance Pages*, on page 233.

Discussion

You could extend this example just a little bit to get a full-fledged system for deploying static web sites. Before you do that, however, look into StaticMatic[77]—it's a nice, compact system that incorporates the concise, powerful, and popular Haml and Sass meta-markup languages to generate static sites.

76. The 422 HTTP code, which stands for "Unprocessable Entity", gets an error page in Rails by default as part of Rails' REST support.
77. http://staticmatic.rubyforge.org/

Part XIII

Big-Picture Recipes

Avoid Starting From Scratch

By Sean Mountcastle (http://seanmountcastle.com)
Sean has been writing software for large telecom and internet infrastructure systems for over a decade. He lives with his wife and children in Northern Virginia. In his spare time, he enjoys family projects, gaming and learning new programming languages.

This recipe was inspired by a post on Josh Susser's blog.[78]

Problem

One of the great features of Rails is the convention for directory layout. Each time you create a new application using the rails command, all of the necessary files and directories are generated for you. But then you have to dress it up with your stuff: installing plugins, copying in useful libraries, static assets, and so on. Wouldn't it be nice if you could save time by creating new Rails applications with all of your customizations already in place?

Solution

It's as easy as maintaining an exemplar Rails project in your version control repository and using it as the cookie-cutter for every new project.

First we create the standard Rails application structure:

```
$ rails exemplar
```

Then we remove those pesky files that we don't need in version control:

```
$ cd exemplar
$ rm log/*
$ rm public/index.html
$ rake tmp:clear
```

The database.yml file generally has database names that are specific to each project, so we'll just store it as database.yml.example:

```
$ mv config/database.yml config/database.yml.example
```

Now that our exemplar application is just the way we like it, it's time to import the directory into our version control repository. Depending

78. http://blog.hasmanythrough.com/2006/12/28/stop-using-the-rails-command

on how you access your Subversion[79] repository—either via HTTP, the SVN protocol, or local disk access—run one of the following commands:

```
$ svn import -m "Initial import" . http://your-svn-server/exemplar/trunk
$ svn import -m "Initial import" . svn://your-svn-server/exemplar/trunk
$ svn import -m "Initial import" . file:///repos/exemplar/trunk
```

When the import finishes, we have all our files safely tucked away in the repository. Next we move the directory we just imported out of the way and check out the exemplar project anew so we have a Subversion-managed local directory[80]:

```
$  mv exemplar exemplar.bak
$  svn co file:///repos/exemplar/trunk exemplar
```

Rails will end up recreating those pesky temporary files for each application, but we can modify the Subversion properties to ignore those files:

```
$ cd exemplar
$ svn propset svn:ignore "*.log" log/
$ svn propset svn:ignore "*" tmp/
$ svn propset svn:ignore "database.yml" config/
```

At this point we have a basic Rails application directory structure checked in to our Subversion repository and a local copy checked out in the exemplar directory. Now it's time to add some of our own goodies.

Everyone uses a plugin (or ten) in all their Rails apps, so let's add one of those to our exemplar using Subversion externals:

```
$ script/plugin -x install svn://errtheblog.com/svn/plugins/will_paginate
```

(We could have also checked in the plugin to our local repository using Piston, as described in Recipe 54, *Managing Plugin Versions*, on page 243.)

We might also have common static assets and Rake tasks, so we'll copy those over to the exemplar, too:

```
$ cp /my/useful/scripts/*.js public/javascripts/
$ cp /my/useful/styles/*.css public/stylesheets/
$ cp /my/useful/errorpages/{404,500}.html public/
$ cp /my/useful/tasks/*.rake lib/tasks/
```

Finally, we commit all these changes and additions back into the repository:

79. I use Subversion, so you'll need to adjust the commands to suit your version control system.

80. We'll access the repository via the local disk from here forward, but the HTTP and SVN protocols work equally well.

Mike says...

I (Heart) Automation

While this recipe may not seem very advanced, it highlights one technique that separates the newbs from the masters: *automation*. A little alarm bell goes off in my head the second time I have to type in a bunch of commands. Experience says it won't be the last time, and I worry that the next time I'll forget to do something (perhaps because I fell asleep at the keyboard from sheer boredom). Automation saves time and, more important, ensures that tasks are done consistently.

```
$ svn commit -m "Updated properties and added common goodies"
```

Here's where the rubber meets the road. When we need to create a new Rails application, we don't use the rails command. Instead, we just make a copy of our exemplar project, like so:

```
$ svn copy -m "Creating my new app"
    file:///repos/exemplar/trunk file:///repos/my_new_app/trunk
```

That makes a copy inside the repository, so we need to check it out to our local my_new_app directory, for example:

```
$ svn co file:///repos/my_new_app/trunk my_new_app
```

This gives us everything we need to get started. Before writin' some code, we need make a few quick project-specific adjustments:

1. Copy config/database.yml.example to config/database.yml
2. Change the database names appropriately for the new application
3. Change the session key in config/environment.rb

Now all that's left is buildin' the application...

Fail Early

By Mike Naberezny (http://maintainable.com)
Mike is the president of Maintainable Software, LLC. He's a member of the Zend Education Advisory Board, regularly speaks at software conferences, and contributes to several open source projects.

Problem

You've no doubt had this experience: You run your application against the wrong migration version and it either blows up in spectacular ways or, worse yet, introduces subtle behavioral changes. It's one thing when this happens on your development box. But when it happens in production, the results can range from embarrassing to downright disastrous.

Solution

Let's corner this migration issue before it has a chance to run amuck—indeed, before our Rails application even starts.

Rails 2.0 introduced the config/initializers directory as a place to store (you guessed it) initialization code without polluting the environment.rb file. Each Ruby file in this directory runs on startup for all environments. So we'll just add a check_migration_version.rb file with the following:

Download buffet/config/initializers/check_migration_version.rb

```
current_version = ActiveRecord::Migrator.current_version rescue 0

highest_version = Dir.glob("#{RAILS_ROOT}/db/migrate/*.rb").map { |f|
  f.match(/(\d+)_.*\.rb$/) ? $1.to_i : 0
}.max

unless defined?(Rake) # skip when run from tasks like rake db:migrate
  if current_version != highest_version
    abort "Expected migration version #{highest_version}, \
           got #{current_version}"
  end
end
```

This code checks that the migration version in our database is the same version as the biggest numbered migration file in our db/migrate directory. If it's not, we're in for trouble. So we fail early with a message to get our attention.

Now imagine that someone on our team checks in a new migration file to the version control repository. We bumble in to the office in the morning and check out all the latest code, but we get interrupted and forget to run rake db:migrate. No worries. When we fire up the application, we're already on top of the problem:

```
$ script/server
=> Booting Mongrel (use 'script/server webrick' to force WEBrick)
...
** Starting Rails with development environment...
Expected migration version 4, got 3
Exiting
```

Things can and do go wrong, so it's better to be safe than sorry, *especially* when dealing with production deployments.

Discussion

This technique works equally well for other application-level invariants. For example, if your app depends on specific MySQL database encodings, adding this initializer gives you an early-warning system:

Download buffet/config/initializers/check_database_encodings.rb

```
DATABASE_ENCODING = "utf8" unless defined? DATABASE_ENCODING

variables = %w(character_set_database
               character_set_client
               character_set_connection)

variables.each do |v|
  ActiveRecord::Base.connection.
    execute("SHOW VARIABLES LIKE '#{v}'").each do |r|
    unless r[1] == DATABASE_ENCODING
      abort "Kindly set your #{r[0]} variable to '#{DATABASE_ENCODING}'."
    end
  end
end
```

Analyzing Your Log Files

By Geoffrey Grosenbach (http://nubyonrails.com)
Geoffrey Grosenbach is the host of the Ruby on Rails Podcast (http://podcast.rubyonrails.org) and producer of PeepCode Screencasts (http://peepcode.com). He blogs at Nuby on Rails (http://nubyonrails.com).

Problem

You want to extract information from your log files for improved analysis. But the standard Rails log format doesn't contain enough data.

Solution

A vanilla Rails production log file might look like this:

```
Processing PeopleController#index (for 74.6.24.207 at 2007-11-05 09:30:44) [GET]
  Session ID: BAh7BzoMY3NyZl9pZCIlNmIxMTY3NDA5YmNlOGIyYzk3ZD
  Parameters: {"action"=>"index", "controller"=>"people"}
Rendering template within layouts/people
Rendering people/index
Completed in 0.03973 (25 reqs/sec) | Rendering: 0.03875 (97%) |
  DB: 0.00027 (0%) | 200 OK [http://yourdomain.com/people]
```

There's enough information there to do rudimentary analysis, but what if we wanted to do performance analysis, or look at what happens at particular times? There's no consistent timestamping of log entries. So the first step is to massage the log file format.

In order to change the format of the log files, we need to make sure we're using a logger that supports customization. Unfortunately, the logger that comes with Rails 2.0 doesn't, so we have to change our environment to force Rails to use the older, more flexible logger:

```
config.logger = RAILS_DEFAULT_LOGGER = Logger.new(config.log_path)
```

This older Logger class lets us specify a custom formatter. We'll see the implementation shortly—for now, let's just see how we tell the logger to use it.

```
require 'recipes_log_formatter'
config.logger.formatter = RecipesLogFormatter.new
```

And that's pretty much it. We'll probably want to change the default log level, so the whole stanza will look like this in our production.rb environment file:

Rails 2.0 Has a New Logger

Back in the old days (prior to Rails 2.0), Rails used Ruby's Logger class for the default Rails logger. Nowadays Rails uses a custom BufferedLogger as the default logger. It's tailored to be as fast as possible. Unfortunately, that means it doesn't support custom log entry formats. Consequently, we need to assign the (old) Logger using config.logger= in this recipe.

Download buffet/config/environments/production.rb

```
require 'recipes_log_formatter'
config.logger = RAILS_DEFAULT_LOGGER = Logger.new(config.log_path)
config.logger.formatter = RecipesLogFormatter.new
config.logger.level = Logger::INFO
```

(Having to set RAILS_DEFAULT_LOGGER as well as config.logger is unfortunate, but is nonetheless required. Without it, script/server won't use our logger, for example.)

Now, we could have simply opened up the Logger class and redefined how log entries are formatted. But the Logger class offers us a way to avoid poking around in its internals. By assigning a custom formatter using the formatter= method, we leave the Logger to its business of logging messages. Then when it's time to format a log entry, the Logger will invoke the call method of its custom formatter.

Which means we need to define our RecipesLogFormatter with a call method:

Download buffet/lib/recipes_log_formatter.rb

```
class RecipesLogFormatter
  def call(severity, time, program_name, message)
    datetime = time.strftime("%b %d %H:%M:%S")
    message      = (String === message ? message : msg.inspect)
    "#{datetime} -- #{message}\n"
  end
end
```

The call method receives four parameters: the log entry's severity, when it occurred, the name of the program it occurred in, and the message to be logged. Our formatter only uses the time and the message, ignoring the other parameters. The message parameter can be any object, so

remember to convert it to a string if it's not already a string (we just used inspect).

Now when we run our application in production, we'll see a timestamp for each log entry in the production.log file:

```
2007-11-05 09:42:46 -- Processing PeopleController#index...
2007-11-05 09:42:46 -- Session ID: BAh7BzoMY3...
2007-11-05 09:42:46 -- Parameters: {"action"=>"index", "controller"=>"people"}
2007-11-05 09:42:46 -- Person Load (0.000286)   SELECT * FROM `people`
2007-11-05 09:42:46 -- Rendering template within layouts/people
2007-11-05 09:42:46 -- Rendering people/index
2007-11-05 09:42:46 -- Completed in 0.04700 (21 reqs/sec) |
  Rendering: 0.04554 (96%) | DB: 0.00029 (0%) | 200 OK
```

So far so good. Things get more exciting when you consider that we can easily modify the call method to format log entries so that they conform to formats recognized by existing log file analyzers. Say, for example, we want to generate log files in the syslog format. [81] Here's an example syslog-compliant log entry:

```
Nov 5 09:14:05 enoch rails[1234]: Is this thing on?
```

In addition to the timestamp and message, we now also need the hostname (enoch is my hostname), the process name (rails), and the process id (1234). To format log entries this way, we just modify our custom formatter slightly. Here's the revised call method:

Download buffet/lib/recipes_log_formatter.rb

```
def call(severity, time, program_name, message)
  datetime = time.strftime("%b %d %H:%M:%S")
  process  = "rails[#{$PID}]"
  hostname = Socket.gethostname.split('.')[0]
  message  = (String === message ?
                message : message.inspect).gsub(/\n/, '').strip
  "#{datetime} #{hostname} #{process}: #{message}\n"
end
```

We ignore the program_name parameter and instead use the process name rails to keep things consistent. And the syslog format is picky about white space, so we tidy up the message. This call method is a tad more involved, and you might want to experiment with it in script/console, for example:

```
$ ruby script/console production
>> RAILS_DEFAULT_LOGGER.error 'Is this thing on?'
```

81. Read more about syslog using man syslog

You'll need to check the production.log file to see if the output is in the correct format. Then when you run your app in production, you should see something like this in the production.log file:

```
Nov 05 09:45:16 enoch rails[1234]: Processing PeopleController#index...
Nov 05 09:45:16 enoch rails[1234]: Session ID: BAh7BzoMY3...
Nov 05 09:45:16 enoch rails[1234]: Parameters: {"action"=>"index"...
Nov 05 09:45:16 enoch rails[1234]: Person Load (0.000277)   SELECT * FROM...
Nov 05 09:45:16 enoch rails[1234]: Rendering template within layouts/people
Nov 05 09:45:16 enoch rails[1234]: Rendering people/index
Nov 05 09:45:16 enoch rails[1234]: Completed in 0.04401 (22 reqs/sec) |
    Rendering: 0.00141 (3%) | DB: 0.00028 (0%) | 200 OK
```

So now that we have log entries in the syslog format, what can we do? Basically we can now run our production log files through any log file analyzer that expects the syslog format. For example, we could install the ProductionLogAnalyzer gem:

```
$ gem install production_log_analyzer --include-dependencies
```

Then to identify potential performance bottlenecks in your application, use the pl_analyze command against your production log file to generate a report:

```
$ pl_analyze log/production.log
Request Times Summary:    Count   Avg     Std Dev Min     Max
ALL REQUESTS:             8       0.011   0.014   0.002   0.045

PeopleController#index:   2       0.013   0.010   0.003   0.023
PeopleController#show:    2       0.002   0.000   0.002   0.003
PeopleController#create:  1       0.005   0.000   0.005   0.005
PeopleController#new:     1       0.045   0.000   0.045   0.045
PeopleController#edit:    1       0.004   0.000   0.004   0.004
PeopleController#update:  1       0.005   0.000   0.005   0.005

Slowest Request Times:
  PeopleController#new took 0.045s
  PeopleController#index took 0.023s
  PeopleController#update took 0.005s
  PeopleController#create took 0.005s
  PeopleController#edit took 0.004s
  PeopleController#index took 0.003s
  PeopleController#show took 0.003s
  PeopleController#show took 0.002s

# DB times and Render times follow
```

To get a performance report on a recurring basis, run the pl_analyze command in a cron job and use the -e option to send the results to an e-mail address. Or generate a performance report on demand from the

comfort of your own computer by running a Capistrano task such as the following:

```
desc "Analyze Rails Log remotely"
task :analyze, :roles => :app do
  run "pl_analyze #{shared_path}/log/#{rails_env}.log" do |ch, st, data|
    print data
  end
end
```

Use pre-defined log file formats or create your own to get the most out of your log files.

Also See

- The Rails Analyzer Tools (http://rubyforge.org/projects/rails-analyzer) includes other handy tools for analyzing Rails log files. For example, the rails_stat command shows a real-time report of application performance by analyzing log files in the syslog format.

Formatting Dates and Times

Problem

You want to customize the format of dates and times displayed in your application. Then, when you inevitably change your mind and need to tweak a format that's used on multiple pages of the application, you want to make the change in only one spot.

Solution

Dates and times are ubiquitous in web applications: when an article was posted, the day you placed an order, how long you'll have to wait for your Wii to ship, and so on. The text you see when you convert a Ruby date or time to a string is usually good enough for us programmers, but often too impersonal for your average web surfer. Sometimes you need to dress up the text a bit, preferably without fiddling around with the strftime method in every case.

Rails already has some pre-defined date and time formats to help get you started. You use them by calling the to_s method on a Date or Time object, passing in the name of the format. As always, script/console is a great way to experiment, so let's go there for some examples. If we want to get the current time in a format that's compatible with our database, we'd use:

```
$ ruby script/console
>> now = Time.now
=> Tue Nov 06 13:48:58 -0700 2007
>> now.to_s(:db)
=> "2007-11-06 13:48:58"
```

Or if we want to show today's date in a long format, we'd use:

```
$ ruby script/console
>> today = Date.today
=> Tue, 06 Nov 2007
>> today.to_s(:long)
=> "November  6, 2007"
```

In addition to :db and :long, Rails includes formats named :short and :rfc822. You can also use the :time format on DateTime objects to just get the time part of the date and time.

If you discover a default format that suits you, go ahead and start using it in your application. At some point though you may find you need to trick out an existing format or, better yet, create a new one altogether. Say, for example, we want to render the time at which an order was placed using a custom format name, like so:

```
<%= order.placed_at.to_s(:chatty) %>
```

And we want the resulting time formatted like so:

```
03:45 PM MST on November 06, 2007
```

To register in our custom format, we need to add the :chatty format to the hash of pre-defined formats. Putting this code in an initializer file ensures it will be automatically loaded. Here's what it looks like:

Download buffet/config/initializers/date_time_formats.rb

```
ActiveSupport::CoreExtensions::Time::Conversions::DATE_FORMATS.merge!(
  :chatty => "%I:%M %p %Z on %B %d, %Y"
)
```

Similarly, we can define custom formats for dates. Suppose we display the date a product will become available in various places around our application. Using a named format means we don't have to remember the formatting details, when we're writing the code or reading it:

```
<%= product.available_on.to_s(:weekday) %>
```

We just need to add the :weekday format to the hash of date conversions:

Download buffet/config/initializers/date_time_formats.rb

```
ActiveSupport::CoreExtensions::Date::Conversions::DATE_FORMATS.merge!(
  :weekday => "%A: %B %d, %Y"
)
```

This gives us the following format:

```
Tuesday: November 06, 2007
```

Even better, we can encapsulate the calls to our formatters inside view helper methods, for example:

```
module ProductsHelper
  def available_on(product)
    product.available_on.to_s(:weekday)
  end
end
```

So now we have more freedom on two axes of change. If we need to change what it means for *any* object to be represented in the weekday

format, we modify the value of that format in our initializer file. If we need to specifically change the format for a product's availability, we modify the view helper method. And that's what being DRY is all about.

Discussion

As a matter of style, consider always defining your own formats that have semantic, rather than format-related, meaning. For example, :short implies "display in this format" whereas in your application using :order_date, for example, is more meaningful.

We didn't go into detail about the strftime[82] method. It has a number of single-character directives for formatting dates and times into strings. A few minutes reviewing the documentation (ri strftime) is, er, time well spent.

82. http://ruby-doc.org/core/classes/Time.html#M000297

Geocoding to Find Things By Location

By Andre Lewis (http://earthcode.com)
Andre Lewis has been working with technology for the last nine years. His experience ranges from large-scale enterprise consulting with Accenture to startup ventures and open source projects. He currently runs his own business, developing Ruby on Rails applications and consulting on Web 2.0 technologies. When he's not working with clients or exploring the latest technologies, he likes to mountain bike, camp, and ride his motorcycle.

Problem

Your application needs to find things based on a physical location. For example, you need to be able to find restaurants up to half a mile from some point, ordered by distance.

Ingredients

- The GeoKit plugin:[83]

```
$ script/plugin install svn://rubyforge.org/var/svn/geokit/trunk
```

- An API key for one or more geocoding web services (instructions included)
- A database that supports trigonometric functions (sorry, SQLite won't cut it)

83. http://geokit.rubyforge.org/

Solution

When you're hungry for sushi, you need to find the nearest sushi bars... and fast! So here's the app we need to build:

Punch in your current location and a radius, and you get a list of sushi bars ordered by distance:

With that goal in mind, we obviously need a way to measure the distance from one street address (where we are) to another (where we could go). It turns out literal street addresses make for lousy math formulas. To accurately calculate the distance between two street addresses, we need their latitude and longitude (lat/lng). Here's the really good news: we don't have to walk the entire surface of earth while holding a GPS unit *or* do any math. Thankfully there are a number of online geocoding services that can handle that for us.

The first step then is to create an account with a geocoding service: Google[84] and Yahoo[85] both provide free geocoding services. For this recipe we just need access to one.

84. http://www.google.com/apis/maps/signup.html
85. http://developer.yahoo.com/wsregapp/

Once you've signed up for an account (don't worry, it's fairly painless), you'll end up getting a key that uniquely identifies your application. We just plug that key into our config/environment.rb file, which the GeoKit plugin updated when it was installed:

```
GeoKit::Geocoders::yahoo = 'REPLACE_WITH_YOUR_YAHOO_KEY'
GeoKit::Geocoders::google = 'REPLACE_WITH_YOUR_GOOGLE_KEY'
```

Now it's time to pump some geocoded data into our database: sushi restaurant names, addresses, and their corresponding latitude and longitude. We'll use scaffolding to create a Restaurant resource for that:

```
$ script/generate scaffold restaurant name:string  ↩
    address:string lat:float lng:float
$ rake db:migrate
```

This gives us everything we need to start creating restaurant data. Except we don't know the latitude and longitude for an address, and even if we did we wouldn't want to be typing them in. No worries—we'll let our geocoding service handle that. We just need to add the acts_as_mappable method with the auto_geocode option to our Restaurant model class:

Download Geocoding/app/models/restaurant.rb

```
class Restaurant < ActiveRecord::Base
  validates_presence_of :lat, :lng

  acts_as_mappable :auto_geocode => true
end
```

OK, let's see if we have everything wired together:

```
$ ruby script/console
>> r = Restaurant.create(:name => "Sushi Den",
                         :address => "1487 S Pearl St, Denver, CO")
=> #<Restaurant id: 9 ...>
>> r.lat
=> 39.689612
>> r.lng
=> -104.98041
```

Hey, look at that! When we created the restaurant record, it automatically got saved with its latitude and longitude values. Let's see how that all worked. The :auto_geocode=>true option to acts_as_mappable added a before_validation_on_create callback method to our Restaurant class. If we wanted to do the same thing manually (or we just wanted more fine-grained control), we could have left off the :auto_geocode=>true option and written this code instead:

```
class Restaurant < ActiveRecord::Base
  validates_presence_of :lat, :lng
  acts_as_mappable
  before_validation_on_create :geocode_address

private

  def geocode_address
    geo = GeoKit::Geocoders::MultiGeocoder.geocode(address)
    errors.add(:address, "Could not geocode address") unless geo.success
    self.lat, self.lng = geo.lat, geo.lng if geo.success
  end
end
```

The workhorse here is the MultiGeocoder class. When we create a new Restaurant object, the value of the address attribute (you can specify a different field name if needed) is transparently sent to whichever geocoding service we've configured. It then sends back the latitude and longitude values for the address, pokes them into our model attributes, and then carries on creating the record.

Now that we've geocoded our restaurant's address and stored its lat/lng values in the database, we're on to our next step: finding sushi bars near us. This turns out to be trivial thanks to location-based finder methods added by the GeoKit plugin. Assuming we've entered a few more restaurants, here's how we find restaurants ordered by distance from a given location, but only up to 5 miles away:

```
>> places = Restaurant.find :all,
     :origin => "9637 East County Line Road, Englewood, CO",
     :within => 5, :order => 'distance'
>> places.size
=> 4
>> places.first.distance
=> "0.84535001110305"
```

It's easy to gloss over what's happening here because it looks just like the ActiveRecord finders we're already used to, with a couple location-specific options. But in fact there's a bit more going on behind the scenes.

First the finder sends off a request to our geocoding service to get the latitude and longitude values for the address string we used as our origin. Then it computes the distance from the origin's lat/lng to the lat/lng of each of our restaurants by running a SQL query representing

a trigonometric formula.[86] In the process of performing the query, a distance field is added to all the Restaurant objects returned by the finder. This represents the distance (in miles, by default) from the origin we used in the query.

If we're just looking for the closest sushi bar (the *I'm Feeling Lucky* of geocoding), we use:

```
Restaurant.find :closest, :origin => "9637 East County Line Road, Englewood, CO"
```

All we need now is a form that asks for the two variables—the address and the radius—and a controller action that runs the query. There's nothing interesting about the form. Here's the controller action:

Download Geocoding/app/controllers/restaurants_controller.rb

```
def search
  @address = params[:address]
  @within  = params[:within]

  @restaurants = Restaurant.find :all,
                    :origin => @address,
                    :within => @within,
                    :order => 'distance'
  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @restaurants }
  end
end
```

To tidy this up, we'd tuck all the options behind a custom finder method that we'd call like this:

```
@restaurants = Restaurant.near(@address, @within)
```

Squirreling away the form parameters in instance variables lets us remind the user what they were searching for on the search results page, just like we had in our opening screenshot:

Download Geocoding/app/views/restaurants/search.html.erb

```
<h1><strong><%= pluralize(@restaurants.size, 'sushi bar') %></strong>
within <strong><%= h @within %> miles</strong>:</h1>

<ul>
<% for restaurant in @restaurants -%>
  <li>
    <p>
```

86. The GeoKit plugin uses the Haversine formula (http://en.wikipedia.org/wiki/Haversine_formula) to calculate the distance between two lat/lng points.

Mike says...

If You Thought This Was Difficult...

Several years and at least one programming language ago, I was contract programming on a web app for a ski/snowboard manufacturer. They didn't have a storefront, but their site let you search for dealers who sold their gear in your area. Back then you needed *all* the geocoded information in your database, and an Excel spreadsheet to figure out how long it would take to implement a dealer locator. These days you can have it done in the time it takes the morning coffee to percolate.

```
      <strong><%= link_to restaurant.name, restaurant %></strong>
      at <%= h restaurant.address %><br/>
      <i>(<%= sprintf("%.2f", restaurant.distance) %> miles away)</i>
    </p>
  </li>
<% end -%>
</ul>
```

Discussion

Sometimes you want to calculate distance in memory, rather than in the context of database queries. To do that, you can call methods of the MultiGeocoder class directly:

```
$ ruby script/console
>> here = MultiGeocoder.geocode("9637 East County Line Road, Englewood, CO")
>> there = MultiGeocoder.geocode("9447 Park Meadows Dr, Lone Tree, CO")
>> here.distance_to(there)
=> 0.845315041284098
```

The MultiGeocoder will use the geocoding services you have configured. If one geocoding service fails to geocode an address, MultiGeocoder will try the next service, in the order you've configured them.

Giving Users Their Own Subdomain

By Mike Mangino (http://www.elevatedrails.com)
Mike Mangino is the founder of Elevated Rails. He lives in Chicago with his wife Jen and their two Samoyeds.

Problem

All the cool sites seem to allow you to have your name or organization in your account URL: mike.famousprogrammers.com, acme.jobpostings.com, and so on. It's the vanity plate of the web. So how do you give *your* users their own URL?

Solution

First we need our Account model to include a subdomain attribute. Here's the minimum migration:

```
create_table :accounts do |t|
  t.string :name, :subdomain
end
```

Then we'll add a handy method to the Account model to determine which account to use for a given subdomain:

```
class Account < ActiveRecord::Base
  def self.for(subdomains)
    find_by_subdomain(subdomains.first) unless subdomains.blank?
  end
end
```

When users access our site by typing in their unique URL, we need to load up their account. Using a before_filter in the AccountsController will do the trick:

Download Subdomains/app/controllers/accounts_controller.rb

```
class AccountsController < ApplicationController

  before_filter :require_account

  def index
    render :text => "<h1>Welcome to your site!</h1>"
  end
```

```
end
```

The require_account method tries to associate the subdomain of the incoming URL with an account, and redirects to the main host if an account can't be found.

```
class ApplicationController < ActionController::Base

  def require_account
    @account = Account.for(request.subdomains)
    if @account.nil?
      redirect_to welcome_url(:host => MAIN_HOST, :port => request.port)
    end
  end
end
```

To test this out, we'll need some domain names that point to our development server. We'll just add entries to our /etc/hosts file.[87] We'll need to define at least a few different hostnames: one for testing subdomains, one for our main page, and one to test a subdomain with no account.

```
127.0.0.1  mike.example.com www.example.com fake.example.com
```

In a production environment, you'll want to set up your DNS server. You should create an A record for your www.example.com domain. Then create a CNAME record for *.example.com that points to www.example.com.

Once we've configured our domain names, we can define the MAIN_HOST constant in our config/environments/development.rb file:

```
MAIN_HOST = "www.example.com"
```

You'll need to set the appropriate URL in your other environment files, as well. For example, you'll use a different MAIN_HOST in development than in production.

Now when we go to http://mike.example.com:3000, we see the welcome URL because we don't have an account. So let's create an account:

```
$ ruby script/console
>> Account.create(:subdomain => "mike")
```

Now if we go to http://mike.example.com:3000 we should see the welcome message for that custom URL. That is, the before_filter found our account and let us into the AccountsController.

So far, so good. Now let's say we don't want to require our users to log in on their own subdomain. Instead, we'll let them login at www.example.com

87. On Windows, the hosts file is in the C:\Windows\System32\Drivers\etc directory.

and then redirect them to their subdomain, such as mike.example.com. To do that, we need to make sure all the cookies will use the example.com domain. That's easy enough—we just add this to the config/environments/production.rb file:

```
ActionController::Base.session_options[:session_domain] = '.example.com'
```

This code forces all cookies to use the example.com domain, and things will work.

Now let's do one better: We'll allow our users to set up a CNAME so that their site can be accessed by a custom domain. For example, they can have their registered customdomain.com domain point to their mydomain.example.com subdomain on our application. To make it work, we need to add a domain column to our Account model and change the Account.for method. Here's the final version of our model:

Download Subdomains/app/models/account.rb

```
class Account < ActiveRecord::Base

  def self.for(domain, subdomains)
    account = find_by_domain(domain)
    unless subdomains.blank?
      account ||= find_by_subdomain(subdomains.first)
    end
    account
  end

end
```

We also need to change the before_filter to shuttle the domain parameter through to the Account.for method:

```
def require_account
  @account = Account.for(request.host, request.subdomains)
  unless @account
    redirect_to welcome_url(:host => MAIN_HOST, :port => request.port)
  end
end
```

Custom domains make for nice branding. Unfortunately, it confuses our sessions because all of our cookies are set up for .example.com. At first blush, it may seem we could just change the cookie domain in our require_account before filter, such as:

```
ActionController::Base.session_options[:session_domain] = '.customdomain.com'
```

Sadly, this won't work. By the time our filter is executed, the outbound session cookies have already been created. Instead, we'll have to poke

around in the CGI library and change the domain of each cookie after the fact.

Download Subdomains/app/controllers/application.rb

```
def set_cookie_domain(domain)
  my_cgi = request.instance_eval "@cgi"
  ocookies = my_cgi.instance_eval("@output_cookies")
  unless ocookies.blank?
    ocookies.each do |cookie|
      cookie.domain = domain
    end
  end
end
```

It's ugly, but we need it to support custom domains as well as subdomains. Now that we have a way to set the cookie domain, we need to change our require_account method again to use it:

Download Subdomains/app/controllers/application.rb

```
def require_account
  @account = Account.for(request.host, request.subdomains)
  if @account
    if request.host == @account.domain
      set_cookie_domain(@account.domain)
    end
  else
    redirect_to welcome_url(:host => MAIN_HOST, :port => request.port)
  end
end
```

Discussion

You'll want to exclude people from registering www as a subdomain, as well as pop, pop3, smtp, mail, ftp, and friends, too. It's also not wise to allow people to register domains under the main name of your app.

If you're going to use SSL, you'll need a wildcard SSL certificate. However, doing so won't completely handle custom domains such as mike.otherurl.com.

Tunneling Back to Your Application

By Chris Haupt (http://www.buildingwebapps.com)
Christopher Haupt is co-founder and Chief Technology Officer at Collective Knowledge Works, Inc. He is a software architect, developer, and educator with over 25 years experience. These days he is focused on applying his experience in ways that can serve the wider development community through efforts such as http://www.buildingwebapps.com/.

Problem

You're developing the next great Facebook (or other) social network application and you don't have the resources (time, spare servers, ability to interrupt existing app, etc.) to constantly redeploy your non-released app for testing. The platform you're deploying to acts as a kind of proxy for your application, so it needs to have public access to your application to operate. Other kinds of web services may have a similar need, calling upon your application via web service APIs. In all cases, your development machine may be behind a firewall or otherwise not accessible to the public Internet.

Ingredients

- A publicly accessible server running the SSH daemon (sshd)
- An SSH client (ssh) on your development machine

Solution

Good ol' Secure Shell (SSH) lets us set up a secure connection (a reverse tunnel) between our development machine and a publicly accessible server. Once the tunnel is open, we can point a social network service such as Facebook to the URL of our public server. Any request made to the public server will be transparently forwarded to our development machine. We can set that up using commands at the terminal, but it's tedious and repetitive. So let's automate it!

First we need to make sure that the SSH daemon (sshd) is set up to allow our public server to act as a gateway. Let's start by logging in via SSH:

```
$ ssh admin@www.example.com
```

Next we locate the sshd_config file (it's /etc/sshd_config on my OS). Within that file, we need to make sure a few important variables are set. Now, be *very careful* here! If you mess up something in this file and restart the sshd process, you'll likely get locked out of your server. So proceed with caution, and double check each variable:

- GatewayPorts needs to be set to clientspecified for recent versions of OpenSSH (4.0 and newer). On others, such as OpenSolaris' default install, it should be set to yes.
- AllowTcpForwarding needs to be set to yes.
- Depending on your configuration, you *may* need to update the KeepAlive variable or the TCPKeepAlive variable to yes. You can change this later if you find your connection drops out frequently. Alternatively, you can add these keep alive settings to your local ~/.ssh/config file:

  ```
  Host www.example.com
  ServerAliveInterval 120
  ```

Once we're done editing the sshd_config file, we need to save it and restart the sshd process. (This is OS-specific, so check your documentation for the right way to do it on your system.)

Next we turn our attention to our local Rails application. To automate setting up the reverse tunnel, we'll create a YML configuration file and create a Rake task. Here's the configuration file:

Download TunnelingBackToYourApp/config/tunnel.yml

```
development:
  public_host_username: admin
  public_host: www.example.com
  public_port: 8868
  local_port: 3000

test:
  public_host_username: admin
  public_host: www.example.com
  public_port: 8868
  local_port: 3000

production:
  public_host_username: admin
  public_host: www.example.com
  public_port: 8868
  local_port: 3000
```

You can choose any public port you wish, just be sure it isn't in use by another program. The local_port is the port you'll run your development Rails environment on. The Rake task just needs to load up the configuration for the current Rails environment and use it to fire up a secure connection:

Download TunnelingBackToYourApp/lib/tasks/tunnel.rake

```
namespace :tunnel do

  desc "Create a reverse tunnel from a public server to a private \
        development server. Use tunnel.yml for parameter configuration."
  task :start => :environment do
    SSH_TUNNEL = YAML.load_file("#{RAILS_ROOT}/config/tunnel.yml")[RAILS_ENV]

    public_host_username = SSH_TUNNEL['public_host_username']
    public_host = SSH_TUNNEL['public_host']
    public_port = SSH_TUNNEL['public_port']
    local_port = SSH_TUNNEL['local_port']

    puts "Starting tunnel #{public_host}:#{public_port} \
          to 0.0.0.0:#{local_port}"
    exec "ssh -nNT -g -R *:#{public_port}:0.0.0.0:#{local_port} \
                          #{public_host_username}@#{public_host}"
  end

end
```

OK, now let's fire it up:

```
$ rake tunnel:start
```

This starts a tunnel in the foreground.

If we now run our Rails app on the correct local port (3000) and hit the public server's URL (http://www.example.com:8868, for example), we should see our app. Once this works, we can then go ahead and set our Facebook application callback URL to http://www.example.com:8868/myapp, for example. Requests will happily flow to our development machine and we won't need to deploy to test changes.

Finally, to kill the connection, simply send an interrupt (CTRL-C) in the terminal.

Discussion

Be sure to enable your Facebook (or other) application to receive traffic from the IPs of both your public servers *and* your development machine. If your development machine (or a router upstream) is supplied with dynamic IPs, you will have to update your Social Networking setting whenever the IP number changes.

Each platform has different set-up techniques for pointing the apps to your tunneled server, but all can use the convenience factor of pointing to a dev box for quick debugging/iterating on code work.

You can reduce the need for entering passwords by setting up SSH keys. There are plenty of good resources on the 'net that explain how to do this for your OS.

You can expand on externalizing this solution for a team by adding the concept of user names to your YAML configuration and selecting on the current username found in the development system's environment variables.

There are various ways to check to see if the tunnel is running, but these tend to be OS specific uses of netstat or other tools. Check your OS doc and a quick Google search will point you down the path of automating that too!

Monitoring (and Repairing) Processes with Monit

Problem

Your application relies on external processes and you need to make sure that all the moving parts continue to, well, move in a well-oiled fashion. Of course you don't want to constantly babysit processes, so you need a way to train the computer to do it for you.

Ingredients

You'll need the Monit[88] utility. Many Linux distributions include Monit and you can use MacPorts to get it for Mac OS X. If all else fails, it's trivial to build from source:

```
$ tar zxvf monit-x.y.z.tar.gz
$ cd monit-x.y.z
$ ./configure
$ make && make install
```

Solution

Monit makes it easy to automate the monitoring and mending of processes. For example, in Recipe 26, *Off-Loading Long-Running Tasks to BackgrounDRb*, on page 133 we fired up a BackgrounDRb server process and then walked away. In production, however, we're wise to employ Monit to periodically check that the process is running, and restart it if something has gone awry.

First we need to write a simple control file to tell Monit what to monitor and how to react to certain conditions. Monit looks for the control file first in ~/.monitrc, then in /etc/monitrc, and finally ./monitrc. Pick your favorite spot.

The control file starts out with a few global settings:

88. http://tildeslash.com/monit/download/

```
set daemon 30
set logfile /path/to/monit.log
set mailserver smtp.example.com
set alert sys-admin@example.com
set httpd port 9111
    allow localhost
```

In this case, Monit will wake up every 30 seconds and check each process (we'll get there), and log status and error messages to our log file. Any unexpected events will be e-mailed to our sysadmin via the SMTP server. The last line starts Monit's built-in HTTP server on port 9111 and makes it accessible only via the localhost, which is useful for checking the status of processes.

Next we define the services we want Monit to keep an eye on. In this case we're just interested in the BackgrounDRb server process, so we only have one service entry. The syntax is quite readable:

```
check process backgroundrb_11006
   with pidfile "/path/to/deploy/current/log/backgroundrb.pid"
   start = "/path/to/deploy/current/script/backgroundrb start"
   stop = "/path/to/deploy/current/script/backgroundrb stop"
   if cpu > 90% for 2 cycles then restart
   if totalmem > 256 MB for 2 cycles then restart
   if 4 restarts within 4 cycles then timeout
   group backgroundrb
```

When the BackgrounDRb server starts up, it drops a process id (PID) file. This is handy because Monit can peek inside the file to determine which process to monitor. If the process has died for some reason, the start directive tells Monit how to fire it back up again. The stop directive is used if we manually stop or restart the process from the command line (more on that later).

OK, so if the process dies, it gets restarted. But other undesirable things can happen too, such as the process going rogue and chewing up precious resources. So Monit gives us a way to test process conditions and take appropriate action early. In this case, if our process exceeds 90% CPU or 256 MB of memory for a duration of 2 Monit check cycles (60 seconds total), then the process will be restarted. And if Monit ends up restarting the process 4 times in a row, then Monit throws up its hands and calls in the humans.

With the configuration file in place, let's run a quick syntax check and then start Monit from the command line:[89]

89. You many want to start Monit automatically after a reboot using init and if the Monit

```
$ monit -t
$ monit
```

At this point it's a good idea to kill the BackgrounDRb server process manually, and watch the Monit log file to make sure it gets started back up on the next cycle.

And that's really all there is to it! We have a fully automated babysitter. No news is good news.

To check the status of our BackgrounDRb server, and all things being monitored, we just use the status command:

```
$ monit status
The monit daemon 4.9 uptime: 25m

Process 'backgroundrb_11006'
  status                            running
  monitoring status                 monitored
  pid                               -1
  parent pid                        -1
  uptime                            32m
  data collected                    Thu Dec 27 12:50:25 2007
```

In addition to monitoring processes, you can also use Monit to start, stop, and restart processes. For example, to restart the BackgrounDRb server process by its name use:

```
$ monit restart backgroundrb_11006
```

Or, if you have multiple BackgrounDRB server processes in the backgroundrb group, you could restart them all using the group name:

```
$ monit restart all -g backgroundrb
```

If you change your control file, you'll need to restart Monit so that it loads the latest configuration:

```
$ monit restart
```

Discussion

This barely scratches the surface of what Monit can do. It's a powerful, and yet easy to use, tool that can monitor processes, files, directories, and even devices on a Unix system. So the next time you're faced with keeping an eye on a resource, look to Monit for automation bliss.

daemon itself dies, init will restart it.

Appendix A

Bibliography

[Fow06] Chad Fowler. *Rails Recipes*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2006.

[HL06] Christian Hellsten and Jarkko Laine. *Beginning Ruby on Rails E-Commerce: From Novice to Professional*. 2006.

[TH05] David Thomas and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2005.

Index

First page of blurb

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Advanced Rails Recipes
http://pragprog.com/titles/fr_arr
Source code from this book, errata, and other resources. Come give us feedback, too!

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

Join the Community
http://pragprog.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://pragprog.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: pragprog.com/titles/fr_arr.

Contact Us

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