Rails 3 Upgrade Handbook

by Jeremy McAnally

Buy the whole book!

Buy the whole book for **$12** and get…

- Almost 120 pages of upgrade information
- A step-by-step guide to upgrading your app to Rails 3
- High-level discussion of what's new in Rails 3
- Practical tips on using Rails 3's new features to improve your code
- Real case studies of upgrading non-trivial applications
- Detailed checklists for upgrading

Head over to http://railsupgradehandbook.com/ for more information and purchasing!

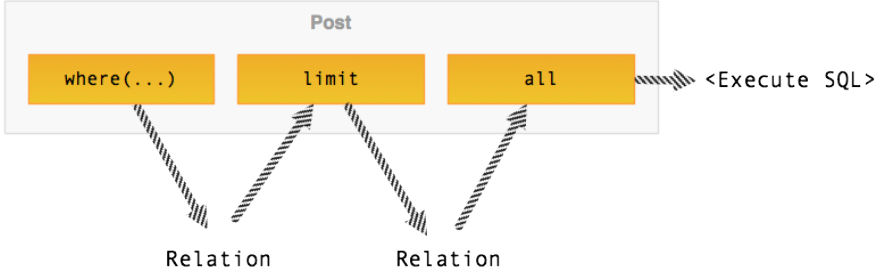
until you execute that query by calling an enumerable method like `all`, `first`, or `each` (you may know this concept as *deferred execution* from other ORMs). So, if you wanted to find the first five `Site` records that are active, you might do this in Rails 2:

```
Site.find(:all, :conditions => {:active => true}, :limit => 5)
```

With Rails 3 and Active Relation, though, the code would look like this:

```
Site.where(:active => true).limit(5)
```

Just like a `named_scope`, these `Relation` objects chain up with each other and compose the SQL query: `Site.where` returns a `Relation` that we call `limit` on, which returns a `Relation` that we call `find` on.



As you'll see in Section 4.4, this new syntax opens up a lot of possibilities that either weren't possible or were quite ugly with the old API.

3. Getting bootable

So, you've prepped the application, got the new files generated, and are now ready to move forward with the upgrade. First, we're going to get the application reconfigured with the new-style configuration files, and then we'll make some essential code changes.

3.1. Configuring your app again

Now comes the task of configuration. There aren't a whole ton of changes, but navigating them can trip up the novice and journey(wo)man alike. Things like initializers and your settings in `database.yml` can generally stay the same; the main changes have to do with `environment.rb`, the router, and gem vendoring.

> **PROTIP:** When using SQLite, the database file is now the `database` key rather than the `dbfile` key.

3.2. Configuring the environment

In all previous Rails versions, most configuration and initialization happened in `config/environment.rb`. In Rails 3, most of this logic has moved to `config/application.rb` and a host of special initializers in `config/initializers`. If you open up `config/environment.rb`, you'll notice it's been seriously slimmed down, and looks like this now:

```
named_scope is now just scope
The named_scope method has been renamed to just scope.
More information: http://github.com/rails/...

The culprits:
        - app/models/group.rb
        - app/models/post.rb
```

It explains the issue, where to get more information on it, and in which files the issue was found. It checks for a wide range of issues (e.g., old generators, busted plugins, environment.rb conversion requirements, old style routes and constants, etc.), so at least run this task, even if you don't use the rest of the plugin; it will save you a lot of time. It doesn't cover everything 100%, but I've found it's great for quickly identifying juicy, low-hanging upgrade fruit.

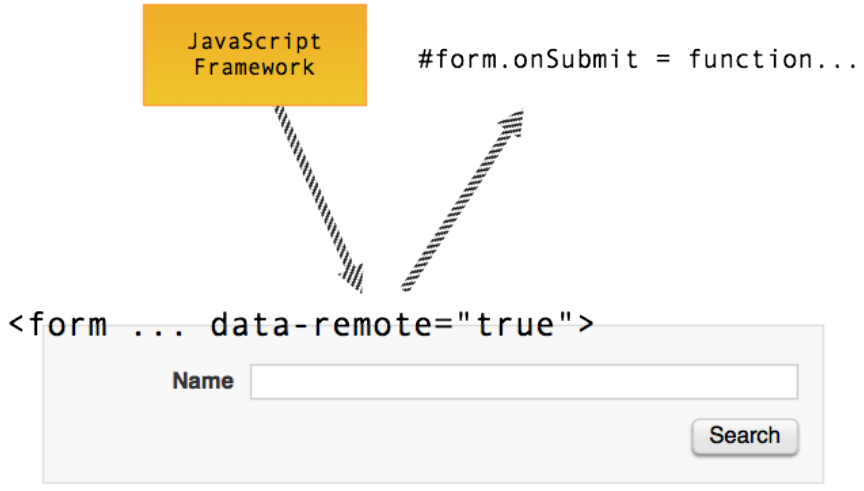
2.2.2. `rails:upgrade:routes`: Upgrading routes

This task will evaluate the routes in your current `routes.rb` file and generate new code for Rails 3. To generate a new routes file, simply run the task inside your Rails application:

```
rake rails:upgrade:routes
```

It will take a routes file like:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :posts, :collection => {:drafts => :get, :published => :get}
```



If you have calls to `link_to_function` or the other helpers, you'd be better served by turning those into proper JavaScript-powered links, writing custom JavaScript to power links sort of like the new Rails helpers do, or downloading the aforementioned plugin to carry you over.

4.3. Building better routes

As you saw in Section 3.2.1, the Rails router DSL has changed significantly, and as part of the refactoring, the Rails core team has also added a number of new features. One of the best new router features in Rails 3 is optional segments; this means that you now have control over what route segments are not required to match the route (whereas before they were hardcoded names like `id`). So, for example, let's say you had an auction site with items that are categorized in categories and subcategories. You might have routes like this in Rails 2.x:

	Deprecation	How to fix it
❑	`config.controller _paths` / `config.controller _paths=` are deprecated.	Use `paths.app. controllers` / `paths.app. controllers=`.
❑	`config.log_path` / `config.log_path=` are deprecated.	Do `paths.log` / `paths.log=` instead.
❑	The `gem` method in application templates deprecated the `:env` key.	Change it to `:only`
❑	The `freeze!` method in application templates is deprecated.	It's no longer needed.

7.2. Plugins

	Deprecation	How to fix it
❑	Putting Rake tasks in `[path]/tasks` or `[path]/rails/ tasks` is no longer supported.	Place them in `[path]/ lib/tasks`.

7.3. Controllers/dispatch

	Deprecation	How to fix it
❑	Disabling sessions for a single controller has been deprecated.	Sessions are lazy loaded, so don't worry about them if you don't use them.
❑	`session.update` is no longer effective.	Use `session.replace` instead.

4.4.3. Caching and relations

One of the smartest uses of the `Relation` objects is to alleviate your caching burden. Normally in a Rails application, you want to cache three things: template rendering, sticky controller logic, and database query results. Rails has facilities for each one, but it's difficult to balance each tier. Often times, people will cache templates but still let their controller logic run; they will cache database objects and forget to properly cache their views.

But with the new Active Record functionality and `Relation` objects, it makes it simple to get a lot of speed out of just a little caching logic. The trick is that queries are not run until you call something like `all` or `each` on the `Relation` object, so if you push this logic into the view as much as possible and cache the view, then you can get twice the caching power from a simple mechanism. So, for example, let's say you had a controller action like this:

```
def index
  @posts = Post.where(:published => true).limit(20)
end
```

Currently, `@posts` is a `Relation` and has not executed a database query. If our view looked like this…

```
<% @posts.each do |post| %>
  <%= render :partial => 'post', :object => post %>
<% end %>
```

…then the query would be executed when we call `each`. But, if we wrap that in a `cache` block like so:

Buy the whole book!

Buy the whole book for **$12** and get…

- Almost 120 pages of upgrade information
- A step-by-step guide to upgrading your app to Rails 3
- High-level discussion of what's new in Rails 3
- Practical tips on using Rails 3's new features to improve your code
- Real case studies of upgrading non-trivial applications
- Detailed checklists for upgrading

Head over to http://railsupgradehandbook.com/ for more information and purchasing!