Rails 4, Part 1: What''s Changed in Rails 4?

January 22, 2013 | By J. Austin Hughey

Note: This is the first in a two-part series that looks at what's changing in Rails 4, and new features in Rails 4. Look out for the second part will be published next week.

The fourth major release of the Ruby on Rails framework is coming up rather soon. While no official release date has been announced, many anticipate a release candidate early this year. This version of the framework has been over a year in the making and represents a major change in the way its internals are architected. The framework has evolved in a more modular format, pushing many of its features into separate gems in an effort to keep the primary codebase lean and mean and remove official support for deprecated features while providing ability to use them if indeed truly needed.

As of the time of this writing, Engine Yard doesn't yet have support for Rails 4 in our Cloud product. That isn't to say that you can't try it - you most certainly can. Be aware that some features, especially live streaming, will probably not work on Cloud right now. Any issues you run into when trying to deploy and run a Rails 4 application should be filed as feature requests instead of bugs since we haven't finished building Rails 4 support into the product yet. We will of course endeavor to have full support for Rails 4 available on our platform as soon as we can.

##Notable Changes in Rails 4

Many things have been changed and/or refactored in Rails 4, and several new features have been added. While this article is not exhaustive, we'll attempt to explain some of the more interesting or impactful ones here.

###Ruby 1.9.3 Minimum

One of the most notable changes in Rails 4 that all developers should be aware of is that it requires Ruby 1.9.3 at a minimum. Support for 1.8.7 has been dropped completely, and since we have 1.9.3 these days, there's really no reason to worry about support for 1.9.2 either. In fact, this commit by Jose Valim specifically tells the user that Rails 4 requires Ruby 1.9.3+ and aborts immediately if RUBY_VERSION is < 1.9.3.

What does this mean for you? First of all, if you're running in 1.8.7 or Ruby Enterprise Edition (which is basically 1.8.7 with some performance tuning capabilities), you should start upgrading now regardless. Upgrading your application from 1.8.7 to 1.9.3 may be fairly painless, or could be rather challenging - it somewhat depends on the application, its dependencies and what (if anything) you were using that may have been deprecated between those releases. Make sure you have good test coverage, switch your interpreter in development mode to 1.9.3 (using RVM makes that wicked simple), and run those tests. Note any unusual output you see and create stories in your favorite tracker system for each item that's incompatible with 1.9.3. For more tips on moving to from 1.8 to 1.9, see this article by Engine Yard's own Anthony Accomazzo.

Regardless, I strongly encourage everyone to run on the latest stable version of the interpreter with the latest patchlevel available for several reasons, performance and security being chief among them.

###No more vendor/plugins

The vendor/plugins directory has been removed in Rails 4. Instead, you should use Bundler with a path or Git dependencies.

###Many deprecated items moved into separate gems

Rails 4 has deprecated many things and moved them into separate gems.

Hash-based and dynamic finder

methods

You may recall that in previous versions of Rails, deprecation warnings for the “old” ActiveRecord query syntax were thrown constantly. For example:

```
User.find(:first, :conditions => { … })
```

Support for this syntax is no longer available in Rails 4. To get this functionality back into your application, you’ll need to install the activerecord-deprecated_finders gem: https://github.com/rails/activerecord-deprecated_finders

ActiveRecord::SessionStore

Storing sessions in the database is an interesting feature that has some useful applications, particularly with respect to sensitive information, but for most cases it’s not as performant as just using a plain ‘ol cookie. So ActiveRecord::SessionStore no longer exists in Rails 4. You’ll need to use the activerecord-session_store gem to bring this functionality back: https://github.com/rails/activerecord-session_store.

To further elaborate, storing sensitive information in a cookie is generally a bad idea for several reasons, lack of encryption (or weak encryption) being only one of them. A better way to go about storing sensitive information is to keep said information in your database (protected by network-layer access and your application), and simply reference object IDs in your session object. So instead of setting a full object in a session, for example, or even meta-data about an object, you may save the object to the database, and simply reference its ID inside the user’s session (and put meta data about it in memcached or PostgreSQL hstore, for example).

ActiveResource

ActiveResource is an ORM for REST-based web services. It’s been removed from Rails 4 and won’t be shipping with it. Guillermo Iguaran says on the Yeti Media blog that the motivation for this change was that it wasn’t being shown the maintenance love that it deserved by the Core team, and that after being extracted, it’s now being given some more attention from its own contributors - a small team led by Jeremy Kemper. There have also been some new features added since the extraction, so all in all, this sounds like a positive change for fans of ActiveResource, and it's quite useful for people who'd like this functionality without necessarily needing to use the entirety of Rails to get it. To get this functionality in your app, include the 'activeresource' gem: https://github.com/rails/activeresource.

Rails Observers, Page and Action Caching

Page and Action caching capabilities are being deprecated in Rails 4 in favor of "Russian Doll" caching strategies. DHH has a gem available on GitHub that explains how that caching will work; see https://github.com/rails/cache_digests for more information on that.

Regardless, with page and action caching now gone, the need for ActiveRecord Observers is now almost non-existent. In many cases, these observers are used to expire caches, which, thanks to Rails 4's new caching scheme, isn't as necessary anymore. Other than cache expiration, observers tend to be abused as a dumping ground for persistence operations in many cases, where a callback is a better option. So Rails 4 will ship without observers, page or action caching.

To get those back in your application, look at:

- https://github.com/rails/actionpack-action_caching
- https://github.com/rails/actionpack-page_caching
- https://github.com/rails/rails-observers

###Dalli instead of memcache-client

Dalli is now being used for memcached instead of memcache-client. This is likely a welcome change in the eyes of many developers, since Dalli is quite a bit faster than memcache-client, and is thread safe - a topic of increasing concern in the community. To use it, you'll need to add the "dalli" gem to your Gemfile if it isn't in there already and set your cache store to mem_cache_store.

###New Default Test Locations

Newcomers to Rails and TDD are often confused by the default setup and naming of test locations. “Models” corresponded to “units”, “controllers” to “functional” tests, and so on. In Rails 4, that’s going to change to be much clearer. The default test locations will now be a little closer to rspec’s naming conventions for Rails tests:

```
app/models -> test/models (was test/units)
app/helpers -> test/helpers (was test/units/helpers)
app/controllers -> test/controllers (was test/functional)
app/mailers -> test/mailers (was test/functional)
```

If you’re ever in the mood to read an interesting discussion on naming of tests and their purposes, you really should check out the discussion around the pull request that implements this change. Originally, the commit was designed to rename “integration” tests (test/integration) to “acceptance” tests. While the rest of the pull request was well received, this particular change sparked a lot of conversation about what an acceptance test is and isn’t. Eventually the committer, blowmage, decided to leave integration as it was so as not to hold up acceptance of the commit, a move well received by several Rails core team members.

###The PATCH Verb

Rails has used the convention of the “PUT” HTTP method for modifying an existing resource for quite a while now. However, the HTTP specification semantically regards a PUT as a complete representation of a resource. In other words, a PUT, semantically-speaking, should contain a full representation of an entire object to be updated. Ergo, technically, we’ve been doing it somewhat “wrong” for a while now by only sending the bits of the object that need to be changed when performing a RESTful update via PUT.

That’s why Rails 4 will change this functionality to use the PATCH verb instead. As it suggests, PATCH is meant to send just new bits of data about an object on the server, not a full representation of it. This is more in-line with HTTP semantics as described in RFC 5789.

As with all RESTful forms in Rails, a hidden “method” field will be generated specifying that the method to be used is “PATCH” instead of “PUT”. And as expected, routing will map the PATCH method to the “update” action in a controller.

In a move to preserve backwards compatibility, the former PUT method will also continue to route to “update” in Rails 4.0, so existing APIs, for example, should continue to work as normal after an upgrade.

The Rails blog goes into more detail on the PATCH method. This is a great change overall, and the fact that it's being done in a backwards-compatible way will certainly smooth the upgrade path for developers.

###Threadsafe by default

Ruby developers have expressed a desire for true, GIL-free multithreading for a while now, and the Rails contributors understand that MRI isn't the only way to run applications in production. Multi-threaded servers and applications allow for significant performance improvements by sharing segments of memory and allowing concurrent execution of code.

However, the standard Ruby interpreter, generally known as "MRI", isn't truly multi-threaded. It can execute code concurrently if one thread is waiting on I/O, for example, and can allow underlying C libraries to execute code concurrently, but actual Ruby code is still executed synchronously due the presence of the global interpreter lock (GIL).

Rails has had the option for running code in a threadsafe manner for a while now, but it hasn't been enabled by default. Starting with Rails 4, it will be. This isn't as major a change as some may expect, given that you can enable this in most Rails apps right now by simply modifying the configuration option and restarting your threadsafe server, though it's worth noting, if for no other reason, than it draws attention to the fact that the Rails community is increasingly adapting to the need to run concurrent web application code.

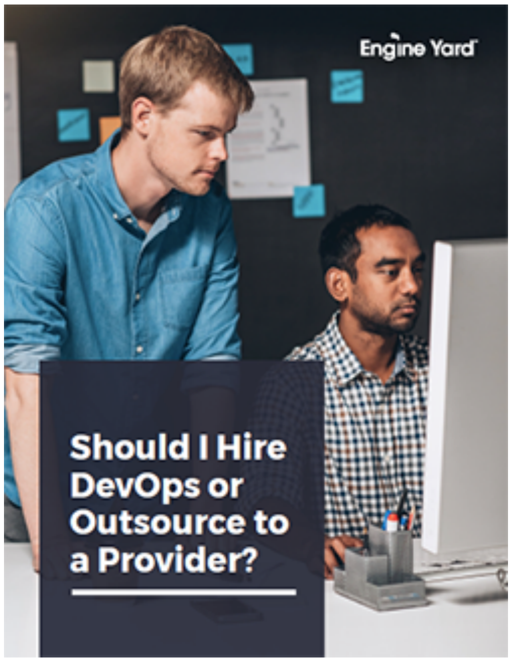
Aaron Patterson goes into more detail on config.threadsafe! here:

http://tenderlovemaking.com/2012/06/18/removing-config-threadsafe.html

That's it for part one. Check back **next week** for part 2, which will cover new features in Rails 4!

Comments

Free Ebook: Should I Hire DevOps or Outsource to a Provider?



You have to invest in your infrastructure: Do you hire DevOps for this critical function, assign it to your already overworked engineers, or outsource to a provider that offers full-stack capabilities?

FIRST NAME *

*First Name**

EMAIL *

*Email**

NUMBER OF SERVERS *

DOES YOUR APP RUN ON RUBY? *

○ Yes
○ No

- Please Select -

☐ Yes, subscribe me to Engine Yard blog post notification emails!

GIMME THE EBOOK

J. Austin Hughey

‹ Prev Post

Next Post ›

Subscribe Here!

Email*

Subscrib

System Status: All Systems Operational

Privacy Policy Leadership Contact

We Hang Out Here Too