Rails 4, Part 2: What''s New in Rails 4?

January 31, 2013 | By J. Austin Hughey

Note: This is the second in a two-part series that looks at what's changing in Rails 4, and new features in Rails 4. Missed the first part? Check it out the first post Rails 4 changes.

Rails 4 also includes some awesome new and useful features that I'm very excited about, and I suspect that you will be, too.

###Postgres H-Store

ActiveRecord in Rails 4 will support the PostgreSQL hstore extension. This database extension allows for a new datatype for a column in PostgreSQL called 'hstore' that effectively represents a string-only hash. For most intents and purposes this is similar to serializing data in a text column, but the fact that this is now a native datatype grants us a very large performance boost and the capability to query against it directly. This means that you can now have a hint of a schema-less database available to your application without needing to perform a full upgrade to MongoDB, CouchDB, Riak or another similar schema-less datastore.

While not a full-blown schema-less solution, the hstore extension and Rails integration with it will be a very welcome addition in many applications. There are times when you may have most data for a particular model well normalized, but may need to allow some information to be specified "on the fly". This is where hstore can truly excel. Let's step through a simple

example. Perhaps your application needs to be able to define new key/value pairs for an object on the fly - defining the schema of an object that will differ from user to user. Maybe this would be, for example, a specific type of form in a CMS. A doctor's office may want to know patient information, or perhaps an event planner may have totally different questions. In the past this may be accomplished using serialization, but with hstore we now have a native way to do this and make it queryable.

You can get started using hstore in PostgreSQL now by using the activerecord-postgres-hstore gem. Ryan Bates also has a Pro Railscast on it that does a great job explaining how it works and shows multiple examples on how to query against it. You can read more about the hstore extension in the official PostgreSQL documentation. It's worth noting that you can use hstore on Engine Yard now by using PostgreSQL 9.1+ as your environment database. To do so, take a look at the postgresql9-extensions custom chef recipes in GitHub. The recipes here will allow you to enable a plethora of extensions beyond just hstore as well, so be sure to look over the various options available to you.

###Array, MACADDR, INET, CIDR

While we're on the subject of new datatypes supported by PostgreSQL and Rails, it's worth noting that ActiveRecord with Rails 4 will have datatype support in PostgreSQL for Array, MACADDR, INET and CIDR datatypes. While these are normally stored as strings in most databases, PostgreSQL supplies native datatype support for each, and ActiveRecord will be able to make use of that.

INET and CIDR datatypes will be converted into instances of the IPAddr class, Arrays will be Arrays of course, and MAC addresses will come out as strings when queried.

###Live Streaming

Aaron Patterson introduced a new feature with Rails 4 called Live Streaming. This effectively makes it possible to stream content to end-user's connected browsers directly from the server via long-polling in the browser. And the fact that the browser is crucial to the process should tip you off to an obvious major problem right away: Internet Explorer doesn't yet support it. This may be fixed in future versions of IE, but for the time being, live streaming is a no-go in Microsoft's browser.

Live Streaming in Rails 4 opens up a lot of very interesting doors in terms of what your application will be able to do. I can see applications relying on real time data being among the biggest beneficiaries: stock tickers and finance applications, gaming, and so on. Other

applications will find uses for live streaming as well.

On Engine Yard, we don't yet support live streaming for a few reasons. First, we have two options for web servers: Unicorn and Passenger. Neither support live streaming yet. Passenger 4 Enterprise, currently in development at Phusion, is planned to support live streaming in Rails, and while we look forward to integrating Passenger 4 into our stack once it's reached release status, we're still working out how - or if - to integrate their Enterprise product into our stack.

This isn't to say that use of Live Streaming is impossible on Engine Yard, just that it isn't possible out of the box at the time of this writing. We plan to change that by opening up more options for application servers. Puma, for example, is currently available as an Early Access feature, as is Rubinius.

###Cache Digests (aka Russian Doll Caching)

If you're tired of bumping version numbers inside nested fragment caching, you're in for a stroke of good luck. Rails 4 will include a cache digest for all fragments cached in your application.

For example, say you have an application caching Authors and Posts:

```
class Author < ActiveRecord::Base
  has_many :posts
end

class Post < ActiveRecord::Base
  belongs_to :author, touch: true
end
```

(Note that adding touch: true to the belongs_to association is crucial since it forces the Author's updated_at to be changed as well when any of an author's Posts are updated.)

Normally you'd probably fragment cache something like this:

```
<!-- app/views/authors/show.html.erb -->
<% cache ['v1', @author] do %>
# <%= @author.name %>'s posts:
<%= render @author.posts %>
<% end %>
```

```
<!-- app/views/posts/_post.html.erb -->
<% cache [‘v1', post] do %>
<%= link_to post.title, post %>
<% end %>
```

The problem is that you have to bump version numbers inside the array every time you want to make a change to your fragment cache. With Rails 4’s cache digest capability, you can simply omit the array and allow an MD5 checksum cache to be generated automatically by Rails. When the file changes, its MD5 checksum changes as well, resulting in a new “cache digest”.

The above can be far better represented with cache digests as:

```
 <!-- app/views/authors/show.html.erb -->
<% cache @author do %>
# <%= @author.name %>'s posts:
<%= render @author.posts %>
<% end %>

<!-- app/views/posts/_post.html.erb -->
<% cache post do %>
<%= link_to post.title, post %>
<% end %>
```

Ryan Bates goes into more detail in this screencast (free), and you can start using this functionality in a Rails 3.2 application today with the cache_digests gem.

Security Related

Rails has always been pretty on-top of things when it comes to security, and this release is no exception. However, in order to take advantage of the latest security enhancements and to keep yourself protected, especially after a vulnerability has been discovered, patched, and disclosed, you should always keep your Rails application up to date. At Engine Yard, we frequently see customers running Rails applications that are several minor point releases behind what’s currently available. For example, why run a Rails 2.3.2 application when you can bump to version 2.3.15 and be protected from known security vulnerabilities?

This is just a little PSA: keep your dependencies up to date by bumping at least the minor point release. Subscribe to the Rails Security mailing list (it's very low volume) and pay attention to these issues, then upgrade your application and test a deployment in a staging environment every time a security issue is disclosed. Otherwise, you're just making yourself a sitting duck for a script kiddie or bot roaming the internet with a database of known vulnerabilities.

###Mass Assignment Protection in the Controller

Personally I'm very happy to see Rails finally move mass assignment protection into the controller in Rails 4. This will be done by essentially whitelisting or blacklisting attributes for an object inside the controller. Let's take a look at a quick example.

Let's say you're building a simple web application to accept recipes (for food, not system configuration). Whoever creates the recipe is known as a "chef" and the recipe has a name, ingredients and directions. We assume here for the sake of expediency that authentication and authorization concerns have already been handled.

```
class RecipesController < ApplicationController
  def create
    @recipe = Recipe.new(recipe_params)
    @recipe.chef = current_user # explicitly assign it to the right user
    if @recipe.valid? and @recipe.save
      redirect_to @recipe,
                  :flash => { :notice =>
                  "Your recipe was added to the cookbook. Mmmm, tasty!" }
    else
      render :action => :new
    end
  end

private

  def recipe_params
    params.require(:recipe).permit(:name, :ingredients, :directions)
  end
```

In this new way of doing mass assignment protection, we tell Recipe.new to accept the hash returned from the recipe_params method, which is specified as a private method in the

controller. This method essentially strips out anything that's not permitted on the "recipe" parameter. So if I were to send the following to this controller (represented by a hash):

```
{name: "Texas Chili", ingredients: "hot stuff", chef_id: "123"}
```

The "chef_id" parameter would be stripped inside the recipe_params method because it's not permitted, and then the "sanitized" hash would be passed to Recipe.new. This would prevent the arbitrary assignment of a Recipe object to any "chef" (author) the attacker wanted.

This is a pretty major change from the way that mass assignment was handled in previous versions of Rails. Instead of putting these concerns inside the model, we see them inside the controller now. This could be somewhat controversial for a number of reasons.

I remember speaking with one of the attendees at the Austin Web Bash about the controversy around where responsibilities for object behavior lie. In purist theory, we should be able to throw any input at an object and trust that the object itself knows what to do with that input. This is how mass assignment protection worked in previous versions of Rails with attr_accessible/attr_protected. You could throw anything at it from any source (tests, live production data via HTTP, console access, etc.) and the object had responsibility for handling it.

However, there's another side to the story. Rails is a web application framework. Which means that you essentially have two input vectors: user, and developer. Developer input, by its very nature, needs to be trusted, whereas user input can never be trusted. Since the only attack vector for the user is in the controller, it seems like that would be the logical place to protect the data instead of inside the model. This opens up more convenience for the developer as s/he works through the data over the lifecycle of a project. In the vast majority of projects, you'll eventually run into situations that require console access or manipulation of data (possibly through migrations). Manually typing out 'object.method = "foo"' for several specific properties every time it's needed is simply a waste of time, mind-numbingly repetitive, and kills productivity. And with mass assignment protection in the model, that's what you're facing. Even in the case of programmatic migrations or tests, you still stand a chance of running into unexpected issues in development and testing due to the mass assignment protection on the model level, even though you're manipulating data in the back-end, in a trusted manner. By putting these concerns in the controller, we get around that problem while maintaining strong user input sanitization.

Essentially, it's a choice: you can have more flexible developer data operations by sanitizing

only controller-based input, or less flexible data operations while maintaining your desired object design. If you want to keep using attr_protected/accessible, you can use the protected_attributes gem in your Rails 4 application.

If you'd like to start using protection on the controller layer now, check out the strong_parameters gem, compatible with Rails 3.2.x.

###Default Headers

Rails 4 will include an option to allow you to specify default headers that are returned with each response from your application. Assuming the browser on the other end properly supports these headers, this can provide the end-user with a much more secure experience when using your application.

In Rails 4, the following headers will be set by default on each response:

```
config.action_dispatch.default_headers = {
'X-Frame-Options' => 'SAMEORIGIN',
'X-XSS-Protection' => '1; mode=block',
'X-Content-Type-Options' => 'nosniff'
}
```

You can set other headers here as well. Read more about this feature in the edge guide on security.

This feature can also be useful to mobile and desktop application developers. You may be able to include some form of metadata by default in all requests with this feature, and then build your desktop/mobile application to look for that meta data. While you can always add headers to each request through controllers, this option allows you the ability to do it automatically on each response.

###Encrypted Cookies

This pull request explains how encrypted cookies will work in Rails 4. In short, you'll have a new session_store option called 'encrypted_cookie_store'. This session store will encrypt cookies as they go out, and decrypt them when they're read by the application. This prevents tampering from the end-user and thus can be an extra layer of security in your application.

Even with this feature, I still have to caution against storing sensitive details in cookies. The

algorithm used to encrypt/decrypt these cookies appears to be SHA-based, meaning that with a cluster of GPUs and some CUDA programming, sensitive data inside the cookie can still be cracked rather quickly since the SHA algorithm family is not as computationally expensive as, say, bcrypt. Not only that, but storing sensitive information client-side is just generally a bad idea. It should be stored in a database of some kind, protected by network-layer access and your application.

Regardless, the capability to encrypt your cookies is very useful for preventing tampering, and will be the default way of storing cookies/sessions in Rails 4.

What's not shipping in Rails 4?

There were a few highly anticipated features of Rails 4 that appear to not be shipping with 4.0. Perhaps they'll make their debut in 4.1.

###Background Queuing

Rails 4 initially included a background queuing API that things like Sidekiq and/or Resque could tie into to do their work. However, this commit moved those capabilities into a separate branch off of master, implying that the feature was somewhat "half baked". Rafael Franca cites that the feature is somewhat a blocker for the release of Rails 4, and postponed its debut so as not to hold up a release.

###Asynchronous ActionMailer

Pursuant to the rollback of the background queuing API, Asynchronous ActionMailer support has also been postponed since it depends on the functionality provided by the background queuing API.

###where.like and where.not_like

Many had anticipated the ability to call a 'not_like' method on a model, e.g. Book.where.not_like(title: "%Java%"), however this commit has rolled back that functionality out of ActiveRecord. DHH lists four reasons why the team decided to roll back the commit,

though not everyone seems to agree. He suggests the community roll this into a gem of their own instead of having it become part of ActiveRecord. Among his arguments against it, DHH mentions that he doesn't like the potential to further push the DSL into things like "greater_than_or_equal", that the use of the percent sign feels like an odd syntactical conflict with Ruby, and that LIKE queries are fairly uncommon in the first place.

Closing Thoughts

Rails 4 has a lot of very interesting new features to offer that will help you build more secure, semantically-correct and efficient applications. I recommend everyone begin an upgrade of their codebase to Rails 4 as soon as it's released, but that you do so in an iterative, orderly fashion.

For those of you still on Rails 2.x, I recommend getting your code stable on at least Rails 2.3.15 (latest release with all security patches) first through a sprint, and then using another sprint or two to move to Rails 3.2. You can migrate release-by-release if you like, just to take things slow and easy, or if you're feeling brave, just push straight to 3.2 and keep weeding out issues along the way as they crop up. Of course, having broad test coverage is going to be key in detecting issues here before they go live, and we always recommend deploying to a staging environment (by cloning your production environment) before pushing anything out to production.

Those of you already on 3.2 will likely have the easiest upgrade path, as much of Rails 4 is a refactor of what already exists with a few new features baked in. Remember to take advantage of the encrypted cookie store and refactor your mass assignment protection to be on the controller level (unless you're really against that sort of thing and prefer to use the aforementioned gem instead).

Overall, Rails 4 is going to be a great release with many excellent refinements on the framework we all know and love. I'm looking forward to it, and I suspect you are, too.

For further study:

- http://blog.remarkablelabs.com/2012/11/rails-4-countdown-to-2013
- http://blog.wyeworks.com/2012/11/13/rails-4-compilation-links/
- http://weblog.rubyonrails.org/2012/11/21/the-people-behind-rails-4/
- http://edgeguides.rubyonrails.org/4_0_release_notes.html

- http://rubyrogues.com/081-rr-rails-4-with-aaron-patterson/
- Rails 4 Whirlwind Tour: http://vimeo.com/51181496

J. Austin Hughey

Comments

‹ Prev Post

Next Post ›

Subscribe Here!

Email*

Subscribe

System Status: All Systems Operational

Privacy Policy Leadership Contact

We Hang Out Here Too