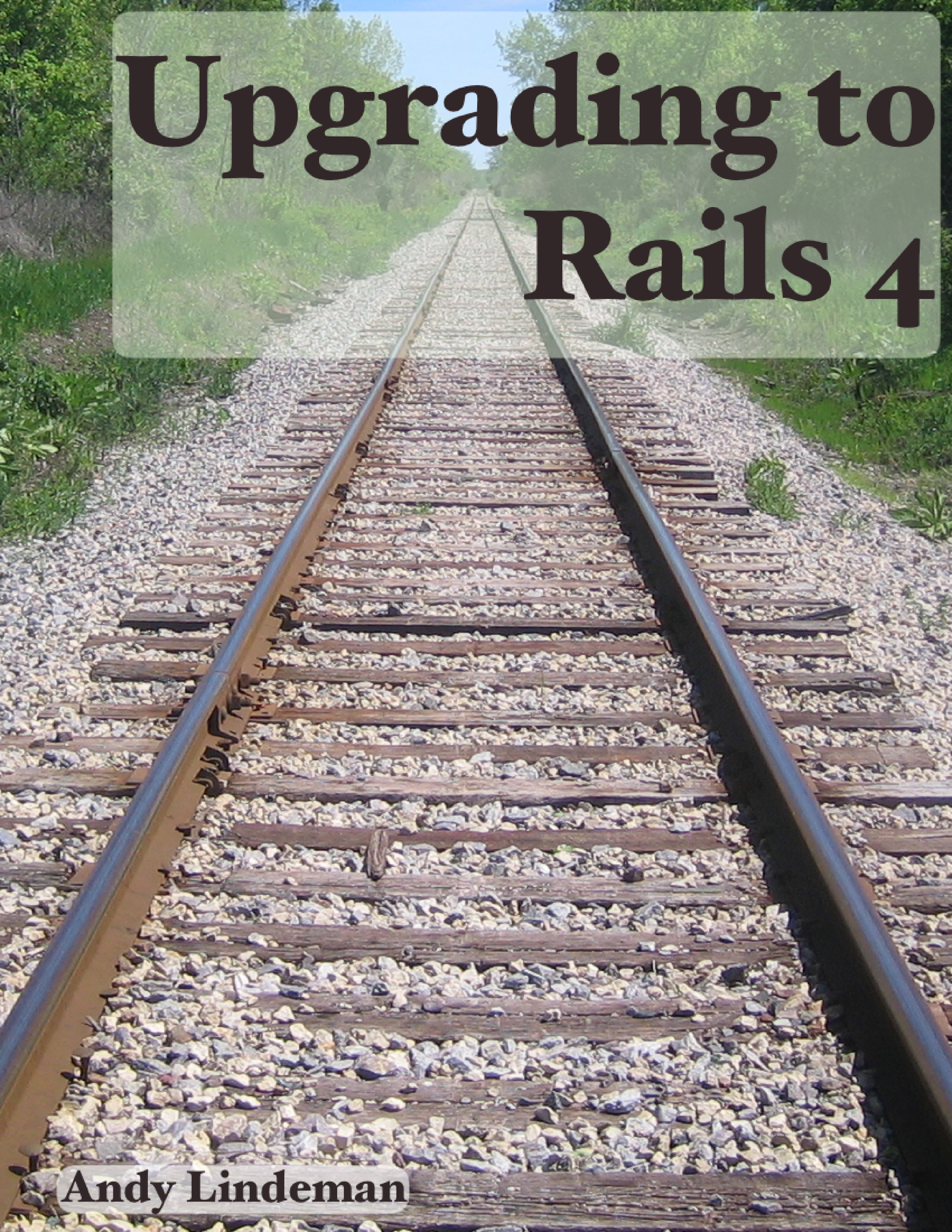
Upgrading to Rails 4



Andy Lindeman

Content

3 Introduction

7 Before You Upgrade

9 Upgrading to Rails 4

15 **Deprecations and Removed Features**

16 Plugins

19 Routing

21 ActiveRecord

31 ActionController and ActionView

39 Asset Pipeline

41 ActiveSupport

42 Caching with memcached

43 Thread Safety

45 ActiveResource

46 Testing

47 **New Features**

48 Routing

50 Model and Controller Concerns

52 ActiveRecord

54 ActionController and ActionView

61 Turbolinks

64 Testing

65 Common Upgrading Scenarios

67 Forking Gems to Loosen Version Constraints

71 **Extracted Gems**

Chapter 1

Introduction

My goal with this book is to provide a clear path for upgrading Rails 3 applications to Rails 4.

The book is split into three major parts:

1. Deprecations: features removed in Rails 4 or marked for removal in a future release
2. New Features: novel features in Rails 4 to be aware of, especially when upgrading
3. Common Upgrading Scenarios: specific issues you might encounter when upgrading a Rails application, and how to fix them

Also included is an Upgrade Checklist which links to important sections that are relevant when upgrading an application. I recommend you read through the Deprecations section entirely first, then use the checklist as a guide when upgrading your applications.

Open Source

The source code for the book is open source hosted on GitHub. You are welcome to submit new content or distribute the book under the terms of the Creative Commons Attribution 3.0 license.

If you find errors or have questions, please reach out by filing an issue.

Website

To purchase additional copies of this book--or a team license--visit http://upgradingtorails4.com.

Cover Photo

The cover photo is copyright Jeramey Jannene and used under the CC BY 2.0 license.

Changelog

1.1.0: October 20, 2013

- The source code for the book is now open source. Proceeds from new sales benefit CFY.

1.0.0: June 25, 2013

- Rails 4.0.0 is released! Updated all of the version numbers accordingly.
- Added information about required configuration changes to enable JavaScript and CSS compression in production.
- Added detailed information on all the gems that were extracted in Rails 4.
- Various copyedits and typo fixes.

0.10.1: April 29, 2013

- Bumps gem versions throughout the book for the release of Rails 4.0.0.rc1.

0.10: April 26, 2013

- Added a Before You Upgrade section highlighting some things to consider before upgrading, and moves the upgrading checklist to this section.
- Updated information about binaries and bundle binstubs, including information about how to upgrade if you are already using binstubs.
- Removed details about the `rails test` command since it has been removed in favor of running tests through `rake` (as is the norm in Rails 3).
- Rails 4 removes the `:assets` group from Gemfile. Be sure to move gems like sass-rails and coffee-rails to the top level.
- Updated information about encrypted cookies. Rails 3 cookies are upgradable to Rails 4 cookies, but the API looks different than it did in Rails 4.0.0.beta1.
- Discussed upgrading RSpec earlier in the book: tests are critical to a smooth upgrade process.
- Noted that using memcached in Rails 4 requires the `dalli` gem.

0.9: March 21, 2013

- Added information about XML parsing being extracted to a gem.
- This handbook assumes that applications being upgraded are running the latest version of Rails 3.2.
- Added information about the `rails test` command which is recommended over `rake test`.
- Recommended raising an error when an unpermitted parameter is received by **strong_paramters**.
- Added details about `validates_confirmation_of` now adding error messages to the _confirmation field.

0.8: February 25, 2013

- Rails 4.0.0.beta1 has shipped! Updated the Upgrading Rails Itself section to use beta1.
- Added information about Ruby 2.0.0, which Rails 4 recommends.
- Added information about concerns, shared pieces of functionality for models and controllers.
- Added information about breaking changes in the asset pipeline.
- Added information about Auto-EXPLAIN for inefficient queries being removed.
- Added information about validates_format_of and the ^ and $ regular expression anchors.

0.7: February 13, 2013

- Added information about the new convention of adding binstubs for `rails`, `rake`, and any other commonly used binaries.
- Added information about performance tests being extracted to a gem.
- Added information about ActiveRecord::Base#update.

0.6: January 7, 2013

- Added a first cut of the upgrade checklist.
- Added information about the changes to testing in Rails 4.
- Removed `journey` from the generated `Gemfile` since `journey` has been integrated into Rails.
- Added information about the extraction of the `encode` option for `mail_to` to the `actionview-encoded_mail_to` gem.
- Noted that routing concerns can be used in Rails 3.2.

0.5: December 25, 2012

- Added information about ActionController::Live and streaming server-sent events.
- Removed `Rails.queue` chapter since it has been punted out of Rails 4. If it returns as a gem or plugin, I will consider adding the material back.
- Updated the Checking for Incompatible Gems section now that Draper 1.0.0 (prerelease) supports Rails 4.
- Added information about declarative ETags.

0.4: December 10, 2012

- Added information about ActiveResource no longer being bundled with Rails.

- Added information about gotchas when using turbolinks.
- Added information about ActiveRecord::Relation#not.
- Added information about JSON serialization in Rails 4.
- Added basic information about routing concerns. I may add some more details later, after I come up with better examples.
- Added information about asynchronous ActionMailer.

0.3: December 3, 2012

- Added information about changes in ActiveSupport.
- Added information about observers being extracted into a gem.

0.2: November 28, 2012

- Clarified that `match` has not been removed, but instead cannot be used without `:via`.
- Added information about encrypted cookies and the encrypted cookie session store.
- Added information about the addition of the PATCH verb.
- Added information about turbolinks.

0.1: November 26, 2012

- First beta release to customers and technical reviewers.

Technical Reviewers

Thank you to these folks for reviewing and giving technical feedback:

- Steve Klabnik
- Caius Durling
- Ashish Dixit
- Ernie Miller
- Darcy Laycock
- Sean Marcia
- Giles Bowkett

Chapter 2

Before You Upgrade

This handbook focuses on upgrading a Rails 3 application to a Rails 4 application.

To use it most effectively, there are a few things you should know first, before you dive into your `Gemfile` and run `bundle update rails`.

Test Suite

Higher level tests, those that drive your application from the outside, become essential during an upgrade process. Any medium to large application needs at least some coverage before an upgrade to Rails 4 is attempted.

If your application has controller tests (also called functional tests) or integration tests (also called request specs or feature specs), you are off to a great start. Run these tests as you go through the upgrade process, to verify breaking changes don't get pushed out to production. If your application lacks these kinds of tests, consider writing some before upgrading.

Checklist

The handbook goes into detail about each major change in Rails 4. I recommend you read through the first section entirely before attempting an upgrade.

Afterward, or after you have upgraded an application or few, you should use this checklist to navigate to specific sections in the order that I expect you will need them during the upgrade process.

1. Upgrade to Ruby 1.9.3 or 2.0.0
2. Upgrade to the latest version of Rails 3.2
3. Upgrade bundler
4. Check for gem incompatibilities using `rails4_upgrade`
5. Upgrade Rails itself
6. Add gems that have extracted functionality from Rails 3
7. Upgrade **rspec-rails** if you use RSpec as your testing framework
8. Add **dalli** if you cache data in memcache
9. Add binaries and binstubs for `rails` and `rake`
10. Upgrade plugins to gems or move code to `lib/`
11. Tweak any routes that use `match` without `:via => :verb`

12. Audit any chained uses of `Relation#order`, as new orders are now prepended rather than appended
13. Change `^` and `$` to `\A` and `\z` (respectively) when using `validates_format_of`
14. Decide whether graceful degradation of remote forms is important to your application and, if so, enable the option to embed authenticity tokens in forms
15. Add any image assets in `lib/` or `vendor/` to the precompilation list

Many configuration options have changed or been removed in Rails 4. You will need to make changes to files like `config/application.rb`, `config/environments/development.rb`, `config/environments/test.rb`, and `config/environments/production.rb`:

1. Remove the `whiny_nils` setting from all environment configuration files
2. Remove the `auto_explain_threshold_in_seconds` setting from all environment configuration files
3. Add new thread-safety configuration options
4. Update JavaScript and CSS compression options

Some functionality from earlier versions of Rails has been deprecated: while your application may continue to operate correctly, you will see warnings. After you have addressed the concerns in the first checklist, consider addressing deprecated features:

1. Modernize Rails 2 finder syntax
2. Modernize dynamic finders
3. Change eager-evaluated scopes to use lambdas
4. Audit any uses of `Relation#all`
5. Address any uses of `Relation#includes` with conditions on the joined table

After your Rails 4 application is running smoothly in production, consider making these changes:

1. Upgrade digitally signed cookies to encrypted cookies
2. Remove any extracted gems that your application does not need

Chapter 3

Upgrading to Rails 4

Since the advent of Bundler, upgrading a Rails application has become much easier.

That said, upgrading to Rails 4 is a bit more involved than simply running `bundle update`.

Rails 3.2

It is much easier to upgrade a Rails application in small steps. If an application you wish to upgrade to Rails 4 is not yet running the latest version of Rails 3.2, you should first upgrade it there as a stepping stone.

In order to provide the best content in a concise form, this handbook assumes applications being upgraded are running on the latest version of Rails 3.2.

Resources that can help upgrade older applications to Rails 3 include:

- Rails 3 Upgrade Handbook
- Rails 3.2 Release Notes

Ruby 1.9.3

Rails 4 *requires* Ruby 1.9.3 and *recommends* Ruby 2.0.0. Attempting to run Rails 4 with a Ruby version below 1.9.3 will cause syntax errors or runtime issues.

While Ruby 2.0.0 is mostly backwards compatible with 1.9.3, I recommend you upgrade only to 1.9.3 during the initial stages of upgrading to Rails 4. You are less likely to run into issues with Ruby itself if the jump is as small as possible. After you have stabilized your application on Ruby 1.9.3 and Rails 4, consider upgrading to Ruby 2.0.0.

Of course, if you are already using Ruby 2.0.0, you're golden.

You can upgrade Ruby with a single command using `rvm`:

```
$ rvm install 1.9.3
```

If an existing application has a `.rvmrc` file in its root directory, it must specify Ruby 1.9.3 or above. A new `.rvmrc` file can be generated by running:

```
$ rvm use --rvmrc 1.9.3
```

If an existing application uses JRuby, Rails 4 requires version JRuby 1.7.0 or above (which runs with Ruby 1.9.3 support by default).

```
$ rvm install jruby-1.7.3
$ rvm use --rvmrc jruby-1.7.3
```

bundler

It is possible that Rails 4 will require a more recent version of `bundler` than is installed in your set of gems.

To avoid any potential problems, simply upgrade to the latest version of `bundler` before going any further:

```
$ gem install bundler
```

rails4_upgrade gem

The `rails4_upgrade gem` helps automate some of the process required to upgrade to Rails 4.

Install the gem in the application that's being upgraded by adding it to `Gemfile`:

```
# Gemfile
gem 'rails4_upgrade'
```

Finish the installation by running `bundle`:

```
$ bundle install
```

Checking for Incompatible Gems

`rails4_upgrade` adds a `rake` task to check for gems that are locked to Rails 3, since they would otherwise prevent an application from upgrading successfully.

Run the task:

```
$ bundle exec rake rails4:check_gems
```

In an ideal world, you would see `"No gem incompatibilities found"`, meaning you could upgrade to Rails 4 straightaway. However, it is more likely that you will be presented with a table of gems and the version of Rails to which they are locked.

In this example, the application depends on a version of `draper` that requires `actionpack` 3.2.x and `activesupport` 3.2.x, both gems in the Rails suite:

```
+-----------------+----------------------+
| Dependency Path | Rails Requirement    |
+-----------------+----------------------+
| draper 0.18.0   | actionpack ~> 3.2    |
| draper 0.18.0   | activesupport ~> 3.2 |
+-----------------+----------------------+
```

Attempting to upgrade to Rails 4 with `draper` 0.18.0 will cause `bundler` to raise an error, as this would violate the constraint set by draper.

If an incompatible gem in the list is under active development, it may already have a version that supports Rails 4. Check the gem's website or source repository (often hosted on GitHub).

If a Rails 4 compatible version is available, specify the compatible version in `Gemfile` and use `bundler` to update it. In this case, `draper` 1.0.0 supports Rails 4.

```
# Gemfile
gem 'draper', '~> 1.0'
```

```
$ bundle update draper
```

If a Rails 4 compatible version is not yet available, it is unfortunately not possible for you to upgrade until that gem adds support or the gem is removed from the application.

Check the gem's issue tracker to see if the authors are aware of the incompatibility; if not, create a new issue.

It is also possible that the gem already works with Rails 4 and the constraints that the gem authors impose are simply unnecessary. Flip to the appendix on forking the gem source and loosening the constraints to see if it is possible to loosen the constraints manually for the time being.

While it may start feeling like a game of whack-a-mole, repeat the process until `rake rails4:check_gems` reports that no incompatibilities exist.

Upgrading Rails Itself

Open `Gemfile` in a text editor and change the line that starts with `gem 'rails'` to:

```
# Gemfile
gem 'rails', '~>4.0.0'
```

Rails 4 also depends on newer versions of gems that drive the asset pipeline (which was introduced in Rails 3.1). Namely, make sure to update `sass-rails` and `coffee-rails` as well:

```
# Gemfile

# Replaces "gem 'sass-rails', '~>3.x.y'"
gem 'sass-rails', '~>4.0.0'

# Replaces "gem 'coffee-rails', '~>3.x.y'"
gem 'coffee-rails', '~>4.0.0'

# Replaces "gem 'uglifier', '~>1.0.3'"
gem 'uglifier', '>=1.3.0'
```

Furthermore, Rails 4 moves away from using an `:assets` group. If you have gems currently grouped in `:assets` in `Gemfile`, promote them to the top level by removing `group :assets`:

```
# Gemfile

# Delete `group :assets` and move these gems
# (and any others) to the top level
gem 'sass-rails', '~>4.0.0'
gem 'coffee-rails', '~>4.0.0'
gem 'uglifier', '>=1.3.0'
```

Finally, Rails 4 moves many features into gems that were previously shipped with Rails itself. Later chapters go into more detail about these changes, and you may not actually need all of these gems for your application. For now, however, add all of the gems that are required to keep existing Rails features working properly after upgrading:

```
# Gemfile
gem 'actionpack-action_caching', '~>1.0.0'
gem 'actionpack-page_caching', '~>1.0.0'
gem 'actionpack-xml_parser', '~>1.0.0'
gem 'actionview-encoded_mail_to', '~>1.0.4'
gem 'activerecord-session_store', '~>0.0.1'
gem 'activeresource', '~>4.0.0.beta1'
gem 'protected_attributes', '~>1.0.1'
gem 'rails-observers', '~>0.1.1'
gem 'rails-perftest', '~>0.0.2'
```

Save `Gemfile` and run from the terminal:

```
$ bundle update rails
```

`bundle` should show output like:

```
Fetching gem metadata from https://rubygems.org/........
Fetching gem metadata from https://rubygems.org/..
Resolving dependencies...
...
```

You're riding on Rails 4!

RSpec

Tests are critical to a smooth upgrade process: they can help highlight breaking changes that need attention with you manually running through each code path in your application.

If you use RSpec as your testing framework, upgrade it to the latest version for Rails 4 support:

```
# Gemfile
group :development, :test do
  gem 'rspec-rails', '>= 2.13.2'
end
```

Finally, run `bundle update rspec-rails`.

Binaries and Binstubs

Binaries--commands like `rails` and `rake`--have been a source of confusion in the Rails ecosystem for a while. For instance, in Rails 3, the `rails` command can be run alone (just `rails`) or via `script/rails`. It can also be run prefixed with `bundle exec`, though it is normally not necessary.

On the other hand, the `rake` command usually *does* need to be prefixed with `bundle exec` (gems like rubygems-bundler remove the need to actually type `bundle exec`, but the effect is simply hidden from view). If you've ever been frustrated by a message claiming "You have already activated rake x.y.z" you know what I mean!

Rails 4 considers binaries to be part of the Rails application itself so they can be invoked in a standard way and versioned alongside the application.

In Rails 4, the `rails` binary now lives in `bin/` and can be invoked as `bin/rails`. Similarly, `rake` can be invoked as `bin/rake`. You are encouraged to check these binaries into version control so that commands run consistently on every machine the codebase is deployed to.

You can generate binaries your application depends on (e.g., `rspec`) with the `bundle binstubs` command. For instance, `bundle binstubs rspec-core` adds the `bin/rspec` command (the `rspec` binary is a part of the `rspec-core` gem).

When upgrading, you should adopt this new convention. To do so, first remove `/bin` from `.gitignore` (if it's currently ignored) so you will be able to add binstubs to version control.

Next, run `bundle exec rake rails:update:bin` to add the `bin/rails` and `bin/rake` binstubs. This is the last time you'll need to use `bundle exec rake`. From now on, you can use `bin/rake` instead.

Better yet, if you add `bin` to your `PATH` environment variable, you can just run `rake`:

```
# ~/.profile
export PATH=$PATH:bin
```

Finally, add these binstubs (and `.gitignore`) to version control:

```
$ git add .gitignore bin/
$ git commit -m 'Adds binstubs for rails and rake'
```

I was already using binstubs in Rails 3!

If you were already using bundler binstubs (generated with `bundle install --binstubs`), Rails will start raising a warning when you use the `rails` command:

```
Looks like your app's ./bin/rails is a stub that was generated by Bundler.

In Rails 4, your app's bin/ directory contains executables that are versioned
like any other source code, rather than stubs that are generated on demand.
```

To upgrade, regenerate the `bundle`, `rails`, and `rake` binaries through Rails (*not* Bundler), and check in those binaries to source control.

Afterward, you can add specific binstubs for commands you want to ship alongside your application.

To get started, run:

```
$ bundle config --delete bin
$ rake rails:update:bin # if asked to override, say 'y'

# Remove bin/ from .gitignore if it appears there
$ git add bin/bundle bin/rails bin/rake
$ git commit -m 'Adds binstubs for rails and rake'
```

Finally, add specific binstubs:

```
$ bundle binstubs rspec-core
$ git add bin/rspec bin/autospec
$ git commit -m 'Adds binstubs for rspec'
```

Deprecations and Removed Features

Chapter 4

Plugins

The term *Rails plugin* can mean multiple things. In this chapter, Rails plugin refers to a piece of shared code that was installed via the `script/plugin install` command and resides in the `vendor/plugins` directory. Applications might have these sorts of plugins even though they fell out of popularity soon after Rails 2.3 was released.

If an application uses gems to manage dependencies, nothing needs to be done. Simply list the contents of the `vendor/plugins` directory in a given Rails application: if the directory is empty or nonexistent, you can safely skip the content in this chapter.

However, if an application is using plugins, you'll need to do a bit of work to successfully upgrade. Read on!

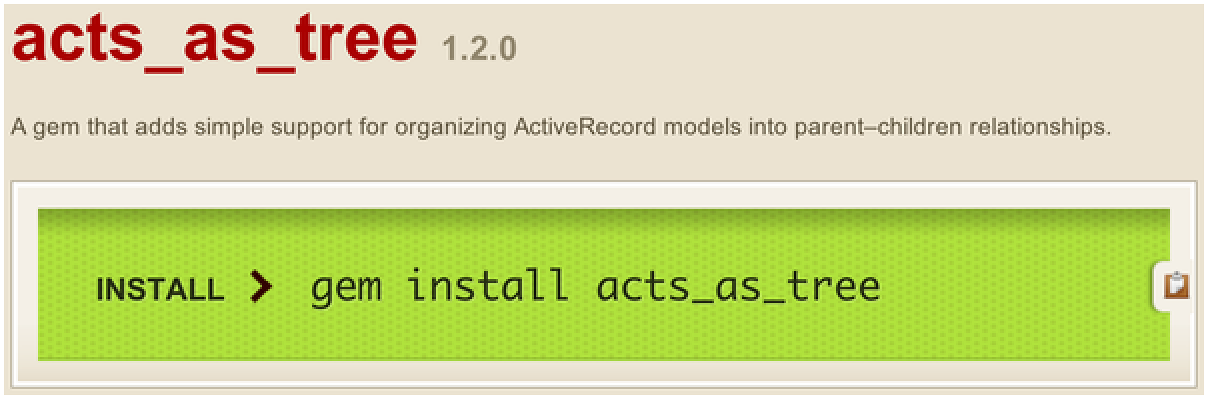
The Easy Option

The easiest way to work around the removal of plugins in Rails 4 is to, well, remove your plugins!

Many popular libraries that once existed as plugins have already been "gemified" by the Ruby and Rails communities. If they have, it may be very easy to replace the plugin with its gem counterpart.

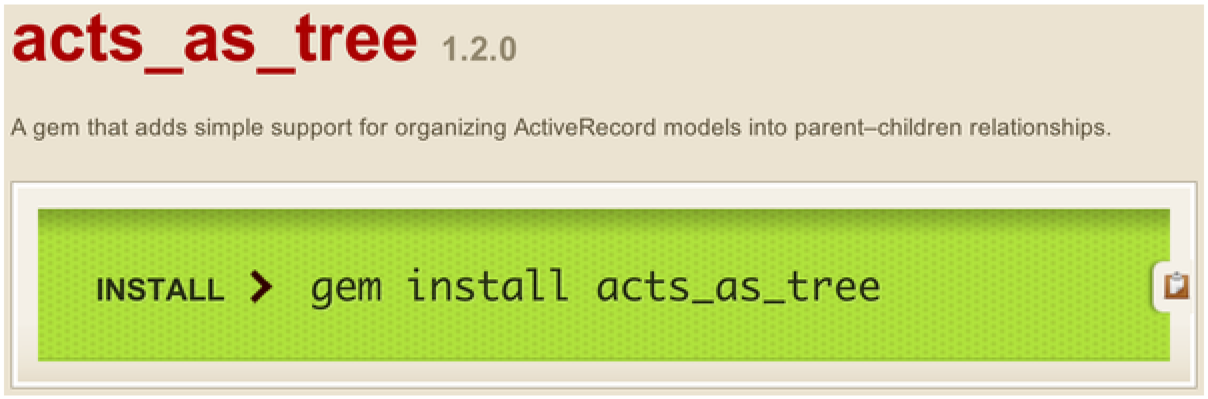
For instance, the `acts_as_tree` plugins is a popular way to model tree structures in ActiveRecord models. Many Rails applications pulled it in as a plugin.

The good news is that `acts_as_tree` has been converted to a gem by open source contributors. The gem's source code is available on GitHub and released on rubygems.org.


acts_as_tree 1.2.0
A gem that adds simple support for organizing ActiveRecord models into parent–children relationships.
INSTALL ❯

```
gem install acts_as_tree
```

To find out if other plugins you need to replace have been released as gems, first search rubygems.org for a gem of the same name as the plugin. If nothing turns up, perform a similar search on GitHub. Look for repositories with descriptions like "Gem version of ...".


acts_as_tree 1.2.0

A gem that adds simple support for organizing ActiveRecord models into parent–children relationships.

```
INSTALL > gem install acts_as_tree
```

The process for replacing a plugin with a gem is straightforward. First, remove the plugin:

```
$ git rm -rf vendor/plugins/acts_as_tree

# If not using git:
$ rm -rf vendor/plugins/acts_as_tree
```

Next, add the gem version of the plugin to `Gemfile`.

```
# Gemfile
gem 'acts_as_tree'
```

If the gem is only available on GitHub, but not rubygems.org, pull the gem in via git instead:

```
# Gemfile
gem 'acts_as_tree', github: 'amerine/acts_as_tree'
```

Install the new gem with Bundler:

```
$ bundle install
```

Finally, wrap your work up in a commit (if using git):

```
$ git add Gemfile Gemfile.lock
$ git commit -m 'Replaced acts_as_tree plugin with gem'
```

Repeat these steps for each plugin. If you run into a situation where you cannot find a gem counterpart for a plugin, read on to the next section.

Adapting Rails Plugins

If the Rails community has not created a gem version of a plugin or the plugin is highly customized for the needs of an application, you must do one of two things:

1. Create a gemified version yourself, or
2. Move the plugin from `vendor/plugins/` to `lib/` and require it manually.

An entire handbook like this one could be written about creating and managing gems, so #1 will not be covered here. It is, however, the more robust solution, and you can check out the following posts to find out more about it:

- How to Gemify your Rails Plugins
- How to convert a Rails plugin into a gem

The #2 option of moving the plugin from `vendor/plugins/` to `lib/` is more straightforward and can serve as a solid temporary solution so that you can proceed with the upgrade.

First, move the plugin's directory:

```
$ git mv vendor/plugins/acts_as_tree lib/

# If not using git:
$ mv vendor/plugins/acts_as_tree lib/
```

Next, add an *initializer* to load the plugin. Save it to `config/initializers/acts_as_tree.rb`:

```
# config/initializers/acts_as_tree.rb
ActiveSupport::Dependencies.autoload_paths <<
  "#{Rails.root}/lib/acts_as_tree/lib"

require "#{Rails.root}/lib/acts_as_tree/init"
```

The first two lines add the plugin's files to the Rails autoload path. Rails will automatically load code from that directory as it's needed. The last line runs the plugin initialization code: since Rails no longer runs it, it must be required manually.

Finally, commit the changes:

```
$ git add config/initializers/acts_as_tree
$ git commit -m 'Loads acts_as_tree in Rails 4'
```

Follow these same steps for each plugin that needs to be adapted.

Chapter 5

Routing

The Rails router is one of the first pieces of code to run when an application receives an HTTP request. Routes determine which controller will handle the request.

Routes are specified in `config/routes.rb`.

match

Rails 4 changes the way the `match` directive operates. `match` was often used incorrectly, and with dangerous consequences.

Consider this route that submits an order to purchase a widget:

```
# config/routes.rb
Widgets::Application.routes.draw do
  match "/widgets/:id/purchase" => "widgets#purchase"
end
```

If this is the route that actually submits an order and charges a user's credit card, the application is vulnerable to the cross-site request forgery (CSRF) attack.

Because `match` without any other qualifiers routes a request with any HTTP verb, including GET, an attacker could lure a user to click a link on a different site that fires a request to purchase a widget:

```
<a href="http://example.com/widgets/1234/purchase">
  Click here! It's awesome! Seriously!
</a>
```

A naive user who clicks the link will have immediately purchased the widget with ID 1234.

Rails has included protection against CSRF attacks for quite some time, but not for requests that use the GET verb. Requests that come in via GET are supposed to be *safe*, meaning they should only be used for retriveal of information, not causing any appreciable server-side state changes. In this case, the application violates that constraint by allowing a GET request to submit a purchase order.

The solution is to update the routes to use a more appropriate HTTP verb, such as POST:

```
# config/routes.rb
Widgets::Application.routes.draw do
  post "/widgets/:id/purchase" => "widgets#purchase"
end
```

In other cases, an application might have been using `match` for *safe* requests. In that case, the directive can simply be updated to `get` to be compatible with Rails 4:

```
Widgets::Application.routes.draw do
  get "/widgets" => "widgets#index"
end
```

You will need to change any instance of `match` in `config/routes.rb` before an application will even boot in Rails 4. If you forget, you will be confronted with an error:

```
You should not use the `match` method in your router without specifying an HTTP
method. (RuntimeError)
```

If you truly need to route a request that comes in via any HTTP verb, modify the `match` route to specify `:via => :any`

```
Widgets::Application.routes.draw do
  match "/something" => "something#index", :via => :any
end
```

Chapter 6

ActiveRecord

Querying the database changed dramatically from Rails 2 to Rails 3. Thankfully Rails 4 does not change things up nearly as much, but there are some improvements and gotchas you need to be aware of.

Rails 2 Finder Syntax

Rails 2 finder syntax is deprecated. If you have an application that uses `find(:all)` and `find(:first)`, you'll need to transition it to the new chained syntax.

```
Post.find(:all, conditions: ["created_at > ?", 2.days.ago])
# DEPRECATION WARNING: Calling #find(:all) is deprecated. Please call #all directly instead.
# You have also used finder options. These are also deprecated.
# Please build a scope instead of using finder options.
```

Rails 4 does not remove the Rails 2 finders, but a future version of Rails might. To be ready, you can squash the deprecation warnings by switching to the chained scope syntax introduced in Rails 3:

```
Post.where("created_at > ?", 2.days.ago)
```

Dynamic Finders

Many of the dynamic finders have been deprecated in favor of alternate syntax. Most of the alternate syntax is already supported in Rails 3.2.

Deprecated Syntax	Preferred Syntax
`find_all_by_...` `scoped_by_...`	`where(...)`
`find_last_by_...`	`where(...).last`
`find_or_initialize_by_...`	`first_or_initialize_by(...)`

`find_or_create_by_...`
`find_or_create_by_...!`
	`first_or_create_by(...)`
	`first_or_create_by!(...)`

An example of each and the newly preferred alternative:

```
# User.find_all_by_last_name("Lindeman")
User.where(last_name: "Lindeman")

# User.find_last_by_email("andy@andylindeman.com")
User.where(email: "andy@andylindeman.com").last

# User.first_or_initialize_by_github_id(395621)
User.first_or_initialize_by(github_id: 395621)

# User.find_or_create_by_twitter_handle("alindeman")
User.first_or_create_by(twitter_handle: "alindeman")
```

Notably, though, the `find_by_...` dynamic finder is *not* deprecated. Code such as `User.find_by_email("andy@andylindeman.com")` will continue functioning without deprecation warnings.

Eager-Evaluated Conditions

scope

Creating a scope without a callable object is deprecated in Rails 4:

```
class Comment < ActiveRecord::Base
  scope :visible, where(visible: true)
end
# DEPRECATION WARNING: Using #scope without passing a callable object is
# deprecated.
```

The preferred syntax in Rails 4 is to create scopes by passing a proc or lambda (though any object that responds to `call` will do).

```
class Comment < ActiveRecord::Base
  # Ruby 1.9 lambda syntax
  scope :visible, -> { where(visible: true) }

  # Also acceptable
  scope :visible, proc { where(visible: true) }
  scope :visible, lambda { where(visible: true) }
end
```

In some situations, eager evaluated scopes are not just deprecated: they will cause errors. For instance, `validates_uniqueness_of` with the `:conditions` option raises an `ArgumentError` if the value is not callable:

```
class Post < ActiveRecord::Base
  # All non-archived titles must be unique
  validates_uniqueness_of :title,
    conditions: where("archived != ?", true)
end
# ArgumentError: ... was passed as :conditions but is not callable.
# Pass a callable instead: `conditions: -> { where(approved: true) }`
```

When upgrading, you must wrap the conditions in a proc or lambda:

```
class Post < ActiveRecord::Base
  # All non-archived titles must be unique
  validates_uniqueness_of :title,
    conditions: -> { where("archived != ?", true) }
end
```

has_many, has_one, belongs_to

Similarly, creating an association with options such as `:conditions`, `:order`, or `:limit` is deprecated in Rails 4:

```
class Post < ActiveRecord::Base
  has_many :recent_comments, class_name: "Comment",
    order: "created_at DESC", limit: 10
end
# DEPRECATION WARNING: The following options in your Post.has_many
# :recent_comments declaration are deprecated: :order,:limit. Please
# use a scope block instead.
```

The preferred syntax in Rails 4 is to create a scope that is merged with the association, wrapped in a proc or lambda:

```
class Post < ActiveRecord::Base
  has_many :recent_comments,
    -> { order("created_at DESC").limit(10) },
    class_name: "Comment"
end
```

In my opinion, the new syntax is much more clear and powerful. That said, upgrading may be painful if you have many associations that use these options.

The full list of deprecated options is shown below. All of these options can be replaced by a scope wrapped in a lambda passed as the second argument to `has\_many`, `has\_one`, or `belongs\_to`:

- `:readonly`
- `:order`
- `:limit`
- `:group`
- `:having`
- `:offset`
- `:select`
- `:uniq`
- `:include`
- `:conditions`

Relation#all

Calling `all` on a relation in Rails 4 will return a new relation instead of an `Array`.

Consider this code snippet and the return values in Rails 3:

```
Post.where("created_at > ?", 2.days.ago).all
# [Post, Post]

Post.where("created_at > ?", 2.days.ago).all.class
# Array
```

In Rails 4, however, `all` does not force the query to an `Array`. It has similar behavior to `scoped` in Rails 3:

```
Post.all.where("created_at > ?", 2.days.ago)
# [Post, Post]

Post.all.where("created_at > ?", 2.days.ago).class
# ActiveRecord::Relation
```

In most cases, the change should not negatively affect your code. Both `ActiveRecord::Relation` and `Array` mix in `Enumerable`, giving them many of the same methods; furthermore, if there is a method that `ActiveRecord::Relation` does not respond to but `Array` does, the method will be proxied through to a version of the results that has been loaded into an `Array`.

However, if you see errors caused by code that previously expected an `Array` and is not handling the change to `ActiveRecord::Relation` properly, you can force the scope to execute and return an `Array` with `to_a`:

```
Post.where("created_at > ?", 2.days.ago).to_a.class
# Array
```

This is also a change you can start making today, as `ActiveRecord::Relation` supports `to_a` in Rails 3.

Relation#includes

The `includes` scope is most often used to eager load associated records, to avoid the N+1 query problem:

```
# OUCH: Executes a query to find all posts, then
# N queries to find each posts' comments!
Post.find_each do |post|
  post.comments.each do |comment|
    # ...
  end
end
```

A solution that `includes` comments allows Rails to run 1 or 2 queries at most:

```
Post.includes(:comments).find_each do |post|
  post.comments.each do |comment|
    # ...
  end
end
```

Rails usually uses an `OUTER JOIN` to perform the eager loading. However, while Rails never *guaranteed* that it would use an `OUTER JOIN`, many developers have written code that takes advantage of this implementation detail.

Consider a query that selects a post and its visible comments:

```
Post.includes(:comments).where("comments.visible = ?", true).find_each do |post|
  # ...
end
```

This will cause a deprecation warning in Rails 4.

```
Post.includes(:comments).where("comments.visible = ?", true)
# DEPRECATION WARNING: It looks like you are eager loading table(s) (one of:
# posts, comments) that are referenced in a string SQL snippet.
```

Rails had to parse the string in the `where` clause to figure out that the `comments` table was referenced.

In Rails 4, you must explicitly tell Rails that the query `references` a joined table:

```
Post.includes(:comments).references(:comments).where("comments.visible = ?", true)
```

Alternatively you can use a hash of conditions, which does not require the `references`:

```
Post.includes(:comments).where(comments: { visible: true })
```

This deprecation will only bite you if you pair `includes` with a `where` condition on the joined table. If you use `includes` only to eager load associations, this deprecation will not affect your code.

Relation#order

Rails 4 changed the way `order` operates when there are multiple calls to `order` in a chain. You might say that, well, Rails 4 changed the order of `order`.

Consider this query that attempts to show comments with many replies first. If comments have the same number of replies, the tie is broken by the creation time of the comment, earlier comments first:

```
Comment.order("replies_count DESC").order(:created_at)
```

This query works correctly in Rails 3. The query generated will be something like:

```
-- Rails 3
SELECT * FROM comments ORDER BY replies_count DESC, created_at
```

In Rails 4, however, the order specifications are flipped: the sort order is `created_at` first and `replies_count` second:

```
-- Rails4
SELECT * FROM comments ORDER BY created_at, replies_count DESC
```

In this case, the query is broken in Rails 4.

One obvious fix is to simply switch the sequence of the `order` calls:

```
# Rails 4
Comment.order(:created_at).order("replies_count DESC")
```

However, a more readable way uses `order` with multiple arguments. This call works correctly in both Rails 3 and Rails 4:

```
# Works in both Rails 3 and Rails 4
Comment.order("replies_count DESC", :created_at)
```

whiny nils

Rails 4 removed *whiny nils*, a feature that would raise a warning when code sent the `id` message to `nil`. Usually this cropped up in applications when code like `@model.id` was run and `@model` was not yet initialized (uninitialized instance variables in Ruby evaluate to `nil`).

Before Ruby 1.9.3, any `Object` would respond to the `id` method, and `nil` is an `Object`. Especially confusing was the fact that `nil.id` returned `4` due to implementation details of Ruby. Ask a developer who has been using Rails for many years about `4` sometime.

Thankfully `Object` instances in Ruby 1.9.3 no longer respond to `id`. Therefore, whiny nils are no longer needed. Attempting to run `nil.id` will simply raise a `NoMethodError`, instead of a confusing warning.

Rails will raise a deprecation warning if an application's configuration attempts to enable whiny nils. To squash the warning, remove any lines that refer to `config.whiny_nils` in `config/environments/development.rb` and `config/environments/test.rb`.

ActiveRecord Session Store

Storing session data in the database using the ActiveRecord session store has been extracted into a gem.

Applications often used the ActiveRecord session store when the data being added to the sessions was sensitive (e.g., users should not be able to decode the contents of the session) or unusually large (over 4KB, the maximum size of a cookie).

To continue using the ActiveRecord session store, bring in the `activerecord-session_store` gem:

```
# Gemfile
gem 'activerecord-session_store', '~>0.0.1'
```

Rails 4 introduces encrypted cookies which may be a good alternative in certain use cases where the ActiveRecord session store was the only option before. I discuss more about encrypted cookies when I talk about the new features in ActionController.

Auto-EXPLAIN Queries

Rails 3.2 added a feature that would automatically run `EXPLAIN` on queries that took longer than a certain amount of time (0.5 seconds by default). Queries that took longer might lack indexes, and `EXPLAIN` would highlight these inefficiencies.

The Rails core team decided that the feature was rarely used in practice, so it is removed in Rails 4. The setting `config.active_record.auto_explain_threshold_in_seconds` should be removed from `config/environments/development.rb` as well as `config/environments/test.rb` and `config/environments/production.rb`.

validates_format_of with ^ and $

Consider a `Post` model with a validation on the format of a URL slug:

```
# app/models/post.rb
class Post < ActiveRecord::Base
  # only alphanumeric chars and the hyphen allowed
  validates_format_of :slug, with: /^[A-Za-z0-9-]+$/
end
```

Unfortunately, the validator does not quite cover all the cases we expected. In Ruby, `^` and `$` match at the beginning and end of a line respectively, but not necessarily at the beginning and end of the entire string.

A sneaky user could submit a post whose slug has multiple lines, and only one of the lines must validate against the regular expression. For example, "thisisvalid\n!@$^&()$@#$#!" passes the validation as written in Rails 3.

Depending on how other parts of the application are written, a validation that uses `^` and `$` could present a security risk because data the developer did not expect to pass the validation. The developer likely meant to use `\A` and `\z` which match at the beginning and end of the entire string respectively.

In Rails 4, you may not use `^` and `$` with `validates_format_of` unless you specifically allow the attribute to be multiline by passing the `multiline: true` option. If you have validations that use `^` and `$` but without `multiline: true`, you will receive an error:

```
The provided regular expression is using multiline anchors (^ or $), which may
present a security risk. Did you mean to use \A and \z, or forgot to add the
:multiline => true option? (ArgumentError)
```

Fixing the issue is straightforward: replace `^` with `\A` and `$` with `\z`:

```
# app/models/post.rb
class Post < ActiveRecord::Base
  # only alphanumeric chars and the hyphen allowed
  validates_format_of :slug, with: /\A[A-Za-z0-9-]+\z/
end
```

This is a positive change because the error is easy to make, yet could have dire consequences for the security of an application.

validates_confirmation_of

Rails 4 changes where error messages are added for fields validated with `validates_confirmation_of` (or `validates :field, confirmation: true`).

A confirmation validation requires that a `field` and its `field\_confirmation` counterpart match.

The typical example is a `User` with a confirmation validation on `password`:

```
# app/models/user.rb
class User < ActiveRecord::Base
  validates :password, confirmation: true
end
```

In Rails 3, a mismatch between `password` and `password_confirmation` will add an error message `"doesn't match confirmation"` to the `password` field:

```
> user = User.new
> user.password = "foo"
> user.password_confirmation = "bar"
> user.valid?
false
> user.errors[:password]
["doesn't match confirmation"]
```

In Rails 4, the error message is added to the `password_confirmation` field instead:

```
> user.valid?
false
> user.errors[:password_confirmation]
["doesn't match Password"]
```

Watch out for this change if you have forms that display error messages inline. You or your team may have designed the form with error messages on `password` in mind, but not `password_confirmation`.

Also take note if your application has been internationalized. Since the error resides on a different attribute, the lookup prefix in the localization file (located in `config/locales`) changes from `activerecord.errors.models.user.attributes.password` and `errors.attributes.password` to `activerecord.errors.models.user.attributes.password_confirmation` and `errors.attributes.password_confirmation` respectively. You will want to rename or retranslate these values if you used them. More information on internationalization of error messages is in the Rails guides.

Observers

Observers have been extracted into a gem. Observers watch ActiveRecord models and are invoked in the same way that callbacks (e.g., `after_save`) are invoked on the model itself.

The Rails guides once said about observers: "Whereas callbacks can pollute a model with code that isn't directly related to its purpose, observers allow you to add the same functionality without changing the code of the model."

In many applications, however, observers cause more problems than they solve. It is often mentally taxing to remember that code in observers ("physically" far away from model code) is run when creating, updating, saving or deleting a record. Gems like no-peeping-toms have been written to disable observers in tests because they can slow the suite down or attempt to interface with external systems.

By extracting observers, Rails 4 is not-so-subtly discouraging their use in new applications.

To continue using observers in upgraded applications, though, simply bring in the `rails-observers` gem:

```
# Gemfile
gem 'rails-observers', '~>0.1.1'
```

JSON Serialization

In Rails 3, HTML entities are output as-is when a model is serialized to JSON. For example, `Post.find(2).to_json` might return a structure like:

```
{
  "id": 2,
  "title": "Hello World",
  "body": "<a href='hello.html'>Hello!</a>"
}
```

However, because the `body` attribute can contain HTML, the application is vulnerable to attacks such as cross-site scripting (XSS) in certain circumstances. If the JSON were embedded directly into a page, for example, an attacker could embed malicious HTML and JavaScript that could steal the user's session or redirect them to another site.

Rails 4 enables the `config.active_support.escape_html_entities_in_json` option by default. The option name is actually a bit confusing: HTML entities are not really *escaped*, but simply encoded differently so they do not evaluate to HTML tags when embedded directly in an HTML page.

In Rails 4, the expression `Post.find(2).to_json` will (by default) return a structure like:

```
{
  "id": 2,
  "title": "Hello World",
  "body": "\u003Ca href='hello.html'\u003EHello!\u003C/a\u003E"
}
```

The angle brackets in `body` have been encoded such that a browser would not interpret them as HTML tags. However, JSON parsers will decode both the Rails 3 and the Rails 4 versions exactly the same. For this reason, it is unlikely that this change will cause you any pain when upgrading. Even so, it is important to understand why the change was made, and why you might see slightly different JSON serialization output between Rails 3 and Rails 4.

Chapter 7

ActionController and ActionView

strong_parameters

Securing a Rails application normally involves protecting against a class of attacks known as mass-assignment vulnerabilities.

The method for doing so has changed in Rails 4.

Consider a `User` model in Rails 3:

```ruby
# app/models/user.rb
class User < ActiveRecord::Base
  attr_accessible :first_name, :last_name
end
```

`User` is protected against mass-assignment vulnerabilities. Specifically, methods that accept a hash of attributes can only set `first_name` and `last_name`.

Next, consider a controller that creates users:

```ruby
# app/controllers/users_controller.rb
class UsersController < ApplicationController
  def create
    @user = User.new(params[:user])
    if @user.save
      redirect_to root_url
    else
      render action: "new"
    end
  end
end
```

The use of `attr_accessible` on the model prevents a client from attempting to set attributes other than `first_name` and `last_name`.

For instance, if a malicious person manipulates the request so that `params[:user] = { admin: "true" }`, the `admin` attribute on the newly created user will *not* be set and the user will *not* become an admin. `User.new` strips out any attributes not specified in the list of `attr_accessible` attributes, protecting the application from mass-assignment vulnerabilities.

However, enforcing mass-assignment protection at the model layer makes models difficult to work with in other parts of the application that do not involve user input and therefore do not need mass-assignment protection. You cannot, for instance, simply create a new admin user at the console with `User.new(admin: true)` even though there is no danger in this situation.

Because of this, Rails 4 adds mass-assignment protection at the controller layer by default via the **strong_parameters** plugin, and these features are available automatically in Rails 4.

In Rails 4, there is no mass-assignment protection in the model by default:

```
# app/models/user.rb
class User < ActiveRecord::Base
end
```

Instead, the controller is responsible for permitting only appropriate attributes through:

```
# app/controllers/users_controller.rb
class UsersController < ApplicationController
  def create
    @user = User.new(user_params)
    if @user.save
      redirect_to root_url
    else
      render action: "new"
    end
  end

  private

  def user_params
    params.require(:user).permit(:first_name, :last_name)
  end
end
```

strong_parameters requires that controllers explicitly slice out attributes that clients are allowed to set before they are passed to the model.

strong_parameters adds two notable methods to `params`:

require(:user)	Requires that `params[:user]` is present (non-empty). If it *is* empty, an "HTTP 400 Bad Request" response is immediately returned.
permit(:first_name, :last_name)	Specifies that only `params[:user]` may only contain the `:first_name` and `:last_name` keys. If other keys are present, they are removed.

Furthermore, Rails 4 will raise an error if a controller attempts to pass `params` to a model method without explicitly permitting attributes via `permit`.

Non-Scalar Values

strong_parameters requires special syntax to permit non-scalar values such as an array or hash.

Consider a system where users have interests such as "programming" or "rugby." A user may have many interests:

```
# app/models/user.rb
class User < ActiveRecord::Base
  has_many :interests
end
```

```
# app/models/interest.rb
class Interest < ActiveRecord::Base
  belongs_to :user
end
```

Users are able to select their interests from a list in a multi-select:

```
# app/views/users/edit.html.erb
<%= form_for @user do |f| %>
  <%= f.label :interests, "Select your interests" %>
  <%= f.collection_select :interest_ids, Interest.all, :id, :name, {}, :multiple => true %>

  <%= f.submit %>
<%- end %>
```

When the form is submitted, the list of interest model IDs will arrive as an array nested inside the `params` hash:

```
# params
{ :interest_ids => ["1", "2", "5"] }
```

To permit an array of IDs with **strong_parameters**, pass a hash key with the name of the attribute and a value of an empty array:

```
# app/controllers/users_controller.rb
class UsersController < ApplicationController
  # ...

  private

  def user_params
    params.require(:user).permit(:interest_ids => [])
  end
end
```

strong_parameters can accept deeply nested arrays and hashes, but needs to be aware of their structure. For in-depth coverage of the syntax, scan the examples in the **strong_parameters** README.

Behavior for Unpermitted Parameters

Any parameter that is not `permitted` is removed from the hash of parameters passed through **strong_parameters**. The model will simply not see it at all.

Furthermore, in the development and test environments, a message is logged to the log file and `rails server` output. However, no error is raised: the request continues.

```
# logs/development.log
Unpermitted parameters: admin
```

I found this lack of an error frustrating in development mode. In my Rails 4 applications, I would sometimes add a new field to the model and to a form, but forget to tweak the `xxx_params` method in the controller.

Submitting the form with the new field would not result in an error, but the new attribute would not be populated either. And, of course, it would take me several minutes to figure out where the problem lay.

I now recommend changing the behavior of **strong_parameters** to raise an exception when an unpermitted parameter is received, though only in development mode. To do so, add a line to `config/environments/development.rb`:

```ruby
# config/environments/development.rb
Widgets::Application.configure do
  # ...

  config.action_controller.action_on_unpermitted_parameters = :raise
end
```

With the configuration option set to `:raise`, it is much more obvious during development when you forget to add a new field to the `permitted` list.

Upgrading

While `attr_accessible` has been removed from Rails 4, it can be brought back via the **protected_attributes** gem.

I recommend adding the **protected_attributes** gem to an application's Gemfile when upgrading:

```ruby
# Gemfile
gem 'protected_attributes', '~>1.0.1'
```

The gem restores `attr_accessible` and disables the requirement that controllers call `params.require.permit` before passing hashes to model methods.

Because **protected_attributes** and **strong_parameters** take very different approaches to enforcing mass-assignment protection, it is not advisable to attempt to use them together in the same application.

I recommend simply using **protected_attributes** when upgrading applications that use `attr_accessible`, and continuing to use `attr_accessible` for mass-assignment protection. At the end of the day, it is your decision as an engineer, but I think in many cases it will be both high risk and low reward to attempt to transition a sizable application from one paradigm to the other at this time.

New applications, however, should use **strong_parameters** and controller-enforced mass-assignment protection, as this appears to be the convention going forward.

Use in Rails 3.2

strong_parameters is one of the many features of Rails 4 that can be used by Rails 3.2 applications today.

I recommend that any new Rails 3.2 applications bring in the **strong_parameters** gem and use it instead of `attr_accessible`.

To get started, simply add **strong_parameters** to the application's Gemfile:

```
# Gemfile
gem 'strong_parameters'
```

Install it via Bundler:

```
$ bundle install
```

Finally, disable the configuration option where Rails will add an implicit `attr_accessible` if one is not specified explicitly:

```
# config/application.rb

# Change from true (default in 3.2.8) to false
config.active_record.whitelist_attributes = false
```

The application is now free to ditch `attr_accessible` in the model layer in favor of `params.require.permit` in the controller layer. The application will already be following the convention for Rails 4 and beyond.

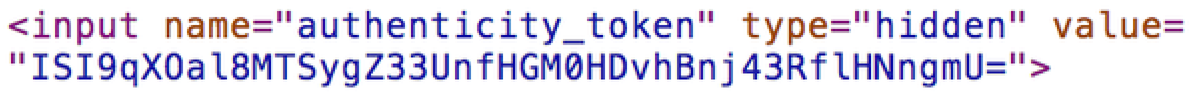
Authenticity Tokens for Remote Forms

Rails protects applications from a range of security issues. By default, Rails requires forms submitted via HTTP verbs other than GET be accompanied with an *authenticity token*.

An *authenticity token* prevents a malicious user from tricking the user's browser into making an authenticated, destructive action from a source other than your application. This attack is called *cross site request forgery* (CSRF).

(As an aside, this is one reason why it is important for all requests that change server state use verbs other than GET.)

Rails automatically embeds an automatically generated authenticity token in forms.



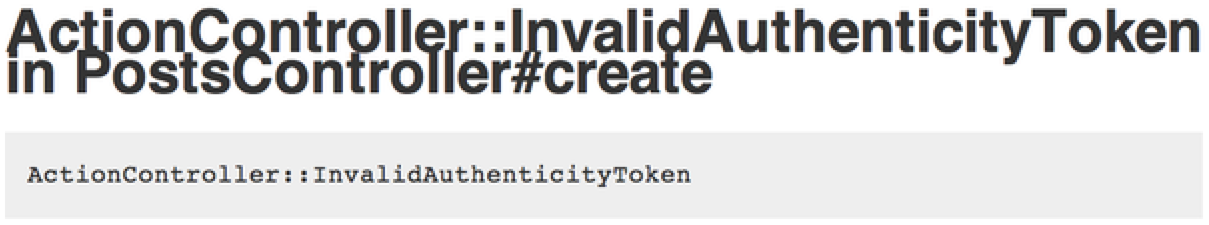
```
<input name="authenticity_token" type="hidden" value=
"ISI9qX0al8MTSygZ33UnfHGM0HDvhBnj43RflHNngmU=">
```

This behavior mostly stays the same in Rails 4; however, **Rails will no longer embed an authenticity token into remote forms submitted via Ajax** (i.e., forms created with `form_for @obj, remote: true`).

Instead, Rails 4 will inject an authenticity token into the request via JavaScript as it is on its way out to the server.

Remote forms submitted in browsers that support JavaScript will be unaffected; however, **remote forms will no longer gracefully degrade if JavaScript is disabled** as they previously did in Rails 3.

Specifically, a remote form submitted in a browser with JavaScript disabled will raise an `InvalidAuthenticityToken` error:



```
ActionController::InvalidAuthenticityToken in PostsController#create

ActionController::InvalidAuthenticityToken
```

While this change may initially appear to have no upsides, forms without embedded authenticity tokens may now be added to fragment caches. Without an authenticity token, the form markup is no longer specific to a user, so it can be cached and reused for all users, speeding up the application.

If your application does not add remote forms to fragment caches, you can address this error and preserve graceful degradation of AJAX forms in Rails 4 by:

- Globally re-enabling embedded tokens by adding `config.action_view.embed_authenticity_token_in_remote_forms = true` in `config/application.rb`, or
- Re-enabling embedded tokens on a case-by-case basis by passing `authenticity_token: true` in the options to `form_for` (e.g., `form_for @obj, remote: true, authenticity_token: true`).

Caching

Rails 4 extracts page caching and action caching to gems.

Page caching saved the response to a request and persisted the data to the filesystem. The next time a request for the same controller action comes in, the response would be served directly by the web server (rather than the Rails application server).

Action caching was similar, but controller filters in the Rails application would still be run. This allowed, for instance, the application to verify that a user had access to the content before it was served.

Both page and action caching are complicated, and often better implemented by out-of-process proxies. In Rails 4, applications that want to continue using page caching and action caching will need to bring in the `actionpack-page_caching` and `actionpack-action_caching` gems respectively by adding them to `Gemfile`:

```
# Gemfile
gem 'actionpack-action_caching', '~>1.0.0'
gem 'actionpack-page_caching', '~>1.0.0'
```

Notably, fragment caching--where smaller pieces of a view are cached--is *not* deprecated in Rails 4. In fact, fragment caching is even improved. More on that in the section describing cache_digests.

XML Parsing

The release of Rails 3.2.11 was prompted in part because a vulnerability in XML parsing allowed the clients to send requests that create symbols within the Ruby environment, as well as evaluate embedded YAML. These and other closely related vulnerabilities turned out to be extremely critical, allowing arbitrary code to be executed in the context of unpatched Rails applications.

These vulnerabilities, combined with the fact that JSON seems to have overshadowed XML as the lingua franca for Rails APIs, prompted the Rails core team to extract XML parsing into a gem in Rails 4.

If your application accepts XML in the request body (note: this is distinct from *rendering* XML as output), you will need to pull in the `actionpack-xml_parser` gem:

```
# Gemfile
gem 'actionpack-xml_parser', '~>1.0.0'
```

If your application does not need to accept XML input, I recommend leaving the gem out in order to reduce the possibility that XML parsing will be the vector for a yet undiscovered security vulnerability.

Applications that simply *render* XML do not need the gem.

actionview-encoded_mail_to

Rails 3 included a little-known feature to obfuscate links to email addresses. This is useful because spambots are known to troll the Internet looking for these `mailto:` links.

The obfuscation was enabled by passing the `encode` option to `mail_to`, usually in a view:

```
# app/views/about/index.html.erb
Contact us <%= mail_to "andy@example.com", "via email", encode: "hex" %>
```

When passed `encode: "hex"`, Rails will obfuscate the email address with HTML entities (in this case `andy@example.com` is transformed into `&#109;&#97;&#105;&#108;&#116;&#111;&#58;%61%6e%64%79@%65%78%61%6d%70%6c%65.%63%6f%6d`).

While a real browser will interpret the encoded email address without any problems, it might trip up a naive spambot.

`encode` can also be set to `"javascript"`. In this case, the email address is encoded into its hex form, but also embedded in a JavaScript snippet that is invoked when the link is clicked. This will likely trip up a higher percentage of spambots, but does now require JavaScript.

In Rails 4, support for `encode` has been extracted to a gem: `actionview-encoded_mail_to`. To enable the feature, simply add the gem to `Gemfile`:

```
# Gemfile
gem 'actionview-encoded_mail_to'
```

Chapter 8

Asset Pipeline

Managing assets like JavaScript, CSS stylesheets, and images was made much easier with the introduction of the asset pipeline in Rails 3.1.

From the perspective of your Rails application, there are not many changes to the asset pipeline in Rails 4. Internally, on the other hand, there are many changes. For example, performance has been dramatically improved: you should notice that asset precompilation during a deploy takes much less time in Rails 4 than in Rails 3.

That said, there are some gotchas you need to be aware of when upgrading.

JavaScript and CSS Compressors

Rails 4 changes the configuration options surrounding compressing assets.

By default, Rails 3 enabled JavaScript and CSS compression with the `config.assets.compress` directive in `config/assets/production.rb`:

```
# config/environments/production.rb
Widgets::Application.configure do
  # Rails 3 setting
  config.assets.compress = true
end
```

Instead, Rails 4 requires you to explicitly specify which compressors to use. In fact, the `config.assets.compress` setting silently has no effect any longer. If you do not update your configuration, assets will no longer be compressed, which could lead to sluggish page loads in production.

When upgrading, remove the `config.assets.compress` directive and replace it with `js_compressor` and `css_compressor`:

```
# config/environments/production.rb
Widgets::Application.configure do
  # Remove config.assets.compress = true
  config.assets.js_compressor  = :uglifier
  config.assets.css_compressor = :sass
end
```

While other compressors can be used, `uglifier` and `sass` are available by default in both Rails 3 and 4.

Precompiled Assets in lib/ and vendor/

Assets (JavaScript, CSS stylesheets, and images) can be added to three directories in a Rails application: `app/assets`, `lib/assets` and `vendor/assets`. Application-specific assets are generally placed in `app/assets`, and assets for third-party libraries are generally placed in `lib/assets` or `vendor/assets`.

Rails 4 no longer automatically precompiles image assets residing in `lib/` and `vendor/` during a deploy. JavaScript and CSS stylesheets in those directories likely *not* affected by this change.

If your application has image assets in `lib/` or `vendor/`, you must either:

1. Move the images to `app/assets/images` (removing them from `lib/assets/images` or `vendor/assets/images`)
2. Add the image filenames to the `config.assets.precompile` list in `config/environments/production.rb` or `config/application.rb`

Consider an image named `sprites.png` that resides in `vendor/assets/images`. In Rails 3, this image would be automatically precompiled. In Rails 4, however, it needs to be added to the list of assets in `config.assets.precompile`:

```
# config/environments/production.rb
Widgets::Application.configure do
  # ...

  config.assets.precompile = %w( sprites.png )
end
```

Frustratingly, `sprites.png` will show up correctly in development regardless of whether it has been added to the precompile list. If it is not added to the list, though, it will break when you deploy to production.

Chapter 9

ActiveSupport

ActiveSupport adds many additional methods on core Ruby objects, and introduces a few new objects of its own. For instance, `"octopus".pluralize` returning `"octopi"` comes from ActiveSupport (specifically, the inflector). The "magic" `params` hash in Rails controllers that allows access to its values via either string *or* symbol keys is `HashWithIndifferentAccess` from ActiveSupport.

Object#try

ActiveSupport adds the `try` method to all `Object` instances. `try` sends a message to an object only if it is not `nil`.

For example, `try` could be used in a view to print the name of a customer's spouse, only if the customer *has* a spouse:

```
<%= @customer.spouse.try(:name) %>
```

If `@customer.spouse` is present (not `nil`), the view will include his or her name. On the other hand, if `customer.spouse` *is* `nil`, the expression itself will return `nil`, and the view simply will not output anything.

Both Rails 3 and Rails 4 react the same way when `try` is called on `nil`:

```
# Both Rails 3 and Rails 4 return `nil`
nil.try(:name)
```

However, Rails 4 changes the behavior when the object receiver is not `nil` yet does not respond to the message being sent:

```
# Rails 4
"abc".try(:bogus_method) # => nil
```

In Rails 3, the same expression will raise a `NoMethodError`:

```
# Rails 3
"abc".try(:bogus_method) # => NoMethodError
```

If you have used `Object#try` and want to keep the Rails 3 behavior when upgrading to Rails 4, switch to the newly-introduced `Object#try!`.

```
# Rails 4
"abc".try!(:bogus_method) # => NoMethodError
```

Interestingly, this same change was made to `Object#try` in Rails 3.1.0 but was reverted in Rails 3.1.1. It seems like it might finally be sticking around for good in Rails 4.

Finally, `Object#try` can no longer be used to invoke private methods in Rails 4. Attempting to call a private method using `try` will simply return `nil` and the method will not be invoked.

Chapter 10

Caching with memcached

Caching data in memcached is a popular way to speed up Rails applications. Rails 4 requires the `dalli` gem, whereas Rails 3 used the now-outdated `memcache-client` gem.

If your application caches data in memcached, you may receive an error tracing to a line in `config/application.rb` or `config/environments/production.rb`:

```
# config/environments/production.rb
Widgets::Application.configure do
  # ...

  config.cache_store = :mem_cache_store, "cache1.example.com"
end
# You don't have dalli installed in your application. Please add it
# to your Gemfile and run bundle install
#
# `rescue in lookup_store': Could not find cache store adapter for
# mem_cache_store (cannot load such file -- dalli) (RuntimeError)
```

The error message sums it up: you need `dalli` instead of `memcache-client`. Adjust Gemfile:

```
# Gemfile

# Remove "gem 'memcache-client'"
gem 'dalli', '~> 2.6.2'
```

Finish up with:

```
$ bundle install
```

Chapter 11

Thread Safety

Traditionally, Rails applications have been deployed on web application servers that only run code on a single thread. In that case, the number of requests an application server can process concurrently is the number of web application server processes that are running.

More and more developers are deploying to threaded web application servers like Rainbows! and puma where a single application server can process concurrent requests without booting the application multiple times in separate processes. Java application servers have always taken advantage of threads, and JRuby can piggyback on their success with servers like trinidad and Torquebox.

---

config.threadsafe! and config.eager_load

Rails 3 required a configuration flag, `config.threadsafe!`, to enable applications to work properly with threaded web application servers. This flag was usually set in `config/environments/production.rb`.

Rails 4 deprecates the `config.threadsafe!` option because Rails applications are now threadsafe by default as long as both `config.cache_classes` and `config.eager_load` are set to `true`.

`config.cache_classes` might seem familiar to you because it exists in Rails 3. It is usually set to `false` in the development environment so that code is reloaded every request as you develop new features. It is normally set to `true` in the test and production environments because reloading code in those environments is not necessary and would only hinder performance.

`config.eager_load` is introduced in Rails 4. When set to `false`, certain application code is not loaded until it is needed. When set to `true`, on the other hand, the entire application is loaded when the application boots.

When upgrading to Rails 4, you will want to add a setting for `config.eager_load` to each of `config/environments/development.rb`, `config/environments/test.rb`, and `config/environments/production.rb`.

Loading the entire application often takes many seconds. For this reason, it makes sense to avoid doing so in the development and test environments. With `config.eager_load` *disabled* in development and test, the entire application does not have to boot if a request or test only needs a subset of the code.

Set `config.eager_load` to `false` in `config/environments/development.rb` and `config/environments/test.rb`:

```
# config/environments/development.rb AND
# config/environments/test.rb
Widgets::Application.configure do
  # ...

  config.eager_load = false
end
```

Unfortunately, loading the entire application upfront is the only way for the application to be thread-safe. Having multiple threads vying to load the same piece of code at the same time is a recipe for disaster.

So, in production, it makes sense to pay the price of eager loading the application in order to be able to run it on a threaded web application server.

Set `config.eager_load` to `true` in `config/environments/production.rb`:

```
# config/environments/production.rb
Widgets::Application.configure do
  # ...

  config.eager_load = true
end
```

With `config.cache_classes` and `config.eager_load` enabled in production, the application is thread-safe and can run on a web application server that uses threads like puma.

For certain applications, this could be a win: it may be possible for you to handle many more requests concurrently without needing more memory. The effect is seen best on Ruby implementations that allow true parallelism like Rubinius or JRuby. Make sure to benchmark and gather metrics on your specific application before switching web application servers in production.

A threaded web application server also makes it feasible to keep long-running connections open to web clients in order to stream data to them over many minutes or even hours. The section on ActionController::Live scratches the surface about what is possible in that realm.

Chapter 12

ActiveResource

ActiveResource no longer ships with Rails by default in Rails 4. ActiveResource allows applications to manipulate remote resources with a syntax similar to ActiveRecord. Usually these resources are exposed by another Rails application using RESTful HTTP API conventions.

ActiveResource has not been well-maintained. Before it was extracted out of the main Rails repository, it was the source of many bugs in the issue tracker. Even so, it did not get much attention from Rails core maintainers.

ActiveResource stills exists in the Rails organization on GitHub, but as of Rails 4, it is independent from Rails' release cycle. For instance, it is possible that ActiveResource 4.0.1 could be released independently of Rails 4.0.1.

Cutting the ties a bit from Rails has already been beneficial for ActiveResource. A new group of maintainers, led by Jeremy Kemper, is working to fix issues and add new features.

ActiveResource 4 adds support for associations (`has_many`, `has_one`, and `belongs_to`). Callbacks (e.g., `after_create` and `after_save`) have also been added. ActiveResource 4 can now be used with APIs that do not follow Rails RESTful conventions. Finally, many bugs have been fixed.

To continue using ActiveResource once you upgrade to Rails 4, add it explicitly in `Gemfile`:

```
# Gemfile
gem 'activeresource', '~>4.0.0.beta1'
```

More information about ActiveResource's new direction and plans for the future are documented in Guillermo Iguaran's blog post ActiveResource is dead, long live ActiveResource.

Chapter 13

Testing

Performance Tests

Rails 3 included a framework for writing performance tests. For example, you might have written a performance test to measure various metrics (wall clock time, memory usage, etc...) of a controller action. Plotting these metrics over time could help pinpoint where a performance degradation was introduced.

Rails 4 no longer includes this framework by default. If your existing Rails application has performance tests, pull in the `rails-perftest` gem:

```
# Gemfile

gem 'rails-perftest'
```

More information about Rails performance tests, including examples, can be found in the README for the rails-perftest project.

New Features

Chapter 14
Routing

PATCH Verb

Rails 4 introduces support for the HTTP PATCH verb. According to RFC 5789, the PATCH verb is appropriate for applying "partial modifications to a resource."

Traditionally, Rails has used the PUT verb for updates of RESTful resources. However, the RFC states that PUT should be used only to "overwrite a resource with a complete new body," and *not* to simply update parts of it.

So, the Rails conventions through Rails 3 have encouraged an incorrect use of PUT. For partial updates, PATCH is the preferred verb.

Rails 4 retains support for update requests coming in via the PUT verb, yet adds support for those same types of requests coming in via the PATCH verb too.

Consider a typical RESTful controller with an `update` action:

```
# app/controllers/widgets_controller.rb
class WidgetsController < ApplicationController
  # ...

  def update
    @widget = Widget.find(params[:id])
    @widget.update_attributes(params[:widget])

    redirect_to @widget
  end
end
```

In Rails 4, requests for `PUT /widgets/1` and `PATCH /widgets/1` will both route to the `WidgetsController#update` action.

You only need to be aware of the PATCH verb; the addition should not cause any upgrade pains as PUT requests continue operating as they always have.

That said, it is recommended that you consider transitioning to using PATCH instead of PUT when your Rails application is an HTTP/RESTful API.

Routing Concerns

Rails 4 introduces **routing concerns**: additions to the routing domain-specific language (DSL) that promise to reduce duplication in certain situations.

Imagine an application that manages many different resources, all of which can be commented on by users. In Rails 3, a route file might look like:

```
# config/routes.rb
Animals::Application.routes.draw do
  resources :dogs do
    resources :comments
  end

  resources :cats do
    resources :comments
  end

  resources :ferrets do
    resources :comments
  end

  # ... etc ...
end
```

Dogs, cats and ferrets have nested comment resources. In Rails 4, this comment "concern" can be extracted and reused:

```
# config/routes.rb
Animals::Application.routes.draw do
  concern :commentable do
    resources :comments
  end

  resources :dogs,    concerns: [:commentable]
  resources :cats,    concerns: [:commentable]
  resources :ferrets, concerns: [:commentable]
  # ... etc ...
end
```

Use in Rails 3.2

While Rails 4 will have routing concerns baked in, Rails 3.2 applications can also use routing concerns by pulling in the **routing_concerns** gem:

```
# Gemfile
gem 'routing_concerns'
```

Chapter 15

Model and Controller Concerns

The Rails community has embraced the concept of skinny controllers, fat models. However, while fat models may be better than fat controllers, fat *anything* is problematic as systems grow.

Rails 4 formalizes the concept of a **concern**: a cross-cutting feature that encapsulates a behavior in the application. DHH writes that Basecamp has over 40 concerns such as "Trashable, Searchable, Visible, Movable, [and] Taggable."

Rails 4 concerns are implemented as Ruby modules and reside in either `app/models/concerns` (for models) or `app/controllers/concerns` (for controllers).

Model Concerns

Imagine a system where records are rarely deleted. Instead, they are merely archived. Archived records are normally hidden from view, but if the evil auditing department ever needs access, these archived records can be dug up.

At first, you might add methods and scopes related to archiving directly to models:

```
# app/models/note.rb
class Note < ActiveRecord::Base
  scope :archived, -> { where(archived: true) }
  scope :unarchived, -> { where(archived: false) }

  def archive
    update(archived: true)
  end

  # ...
end
```

After many models accumulate this same archiving code, you might extract it into a shared place: a model concern. Concerns are just Ruby modules, though they normally extend `ActiveSupport::Concern` for some added convenience.

The previous code could be extracted into `Archivable`, stored in `app/models/concerns/archivable.rb`:

```ruby
# app/models/concerns/archivable.rb
module Archivable
  extend ActiveSupport::Concern

  included do
    scope :archived, -> { where(archived: true) }
    scope :unarchived, -> { where(archived: false) }
  end

  def archive
    update(archived: true)
  end
end
```

Finally, models that can be archived simply include the `Archivable` module:

```ruby
# app/models/note.rb
class Note < ActiveRecord::Base
  include Archivable
end
```

```ruby
# app/models/project.rb
class Project < ActiveRecord::Base
  include Archivable
end
```

Controller Concerns

Controller concerns are modules of shared behavior applied to controllers.

For example, consider a `RequiresAuthentication` concern that protects certain controllers from unauthenticated access:

```ruby
# app/controllers/concerns/requires_authentication.rb
module RequiresAuthentication
  extend ActiveSupport::Concern

  included do
    before_filter :require_authentication
  end

  def require_authentication
    @current_user = User.find_by_id(session[:user_id])
    redirect_to login_url unless @current_user
  end
end
```

To use, simply include in any relevant controllers:

```ruby
# app/controllers/super_secret_controller.rb
class SuperSecretController < ApplicationController
  include RequiresAuthentication

  # ...
end
```

Chapter 16

ActiveRecord

Rails 4 did not just deprecate features of ActiveRecord; it also added some compelling new ones.

Relation#none

Consider an authorization scheme where only approved users can view posts.

Before Rails 4, writing an `authorized` method on `Post` was difficult because there is no reliable way to create a scope that will never return anything. An option that was often used was to return an empty array:

```
class Post < ActiveRecord::Base
  def self.authorized(user)
    user.approved? ? all : []
  end
end
```

However, this is problematic because any code that uses the `authorized` method must know that it could potentially return an array, and in that case, no more scopes can be chained. Consider a controller that uses the method:

```
class PostsController < ApplicationController
  def index
    @posts = Post.authorized(current_user)
    unless @posts.is_a?(Array)
      @posts = @posts.limit(10)
    end
  end
end
```

This is a contrived example: the astute reader will recognize that the `unless` conditional can be removed if `authorized` is the last method in the chain. However, a more complicated chain of methods and scopes might not be able to be simplified in this way.

In any case, the code is too surprising: the `authorized` method has two very different return values: one where more scopes *can* be added, and the other (an `Array`) where more scopes *cannot* be added.

All is not lost! Rails 4 introduces the `none` scope. The `authorized` methods can now be written as:

```
class Post < ActiveRecord::Base
  def self.authorized(user)
    user.approved? ? all : none
  end
end
```

A controller can now query for posts that the current user is authorized to see, but not worry that it may be "none":

```
class PostsController < ApplicationController
  def index
    @posts = Post.authorized(current_user).limit(10)
  end
end
```

The `none` scope is returned by `Post.authorized` if the user is not yet approved. In that case the query will never return any posts, but the controller can tack on more scopes (i.e., `limit(10)`) even so.

The `none` scope is implemented by returning an `ActiveRecord::NullRelation`, named for the null object pattern. No database query will be used when a `none` scope is used in the chain.

Relation#not

Imagine an application has a `Comment` model, and you wish to find all of the comments authored by users *other* than the current user.

In Rails 3, writing a query that needs the not-equal-to operator (`!=`) requires using `where` with a string condition:

```
Comment.where("user_id != ?", current_user.id)
```

Rails 4 introduces the `not` scope. In Rails 4, the same query could be rewritten as:

```
Comment.where.not(user_id: current_user.id)
```

Notably, no string clause is required. The result is a cleaner expression that is guaranteed to operate correctly regardless of the underlying database.

ActiveRecord::Base#update

Rails 4 introduces `ActiveRecord::Base#update`, allowing you to use `update` instead of `update_attributes` when updating the attributes of an ActiveRecord model.

`update` is more terse and matches the name of the controller action where it is normally used, a property that `new`, `create`, and `destroy` have shared for a while now.

```
comment = Comment.find(1)

# Rails 3
comment.update_attributes(body: "Updated content")

# Rails 4 (same effect)
comment.update(body: "Updated content")
```

`update` and `update_attributes` can be used interchangeably in Rails 4. In fact, `update_attributes` is only "soft deprecated": unlike other deprecations, it will not raise any visible warnings. While `update_attributes` may be removed in future versions of Rails, it will probably not be anytime soon.

Chapter 17

ActionController and ActionView

ActionController::Live

By default, Rails renders the entire view template and layout before sending any data back to a browser. While simple, this approach adds to the critical path of time it takes before the resulting page appears to the user.

Rails 3.1 introduced optional streaming templates: when enabled, parts of the view (e.g., the layout) are rendered and sent back to users's browser before the *entire* view is rendered. Importantly, the browser can start downloading and parsing assets like JavaScript, stylesheets, and images while the rest of the view is being generated. More information about this kind of streaming is available in the `ActionController::Streaming` API documentation.

However, `ActionController::Streaming` does not give developers full control over the streaming process and is geared mostly toward speeding up short-lived requests. Recent developments in HTML5 and JavaScript make it more appealing to keep long-lived connections open between servers and browsers. To achieve that alongside Rails, though, many developers have been forced to use a separate Sinatra application or even reach for technologies like node.js.

Rails 4 introduces `ActionController::Live` which gives developers full control over sending arbitrary data to browsers, including over long-held connections.

An example is the easiest way to explain this new feature in detail.

As a prerequisite, a Rails application using `ActionController::Live` should be configured for thread safety. Rails 4 makes this straightforward, as I explain in the Thread Safety section earlier. Thread safety is already the default in production, but the development configuration needs to be adjusted by editing `config/environments/development.rb`:

```
# config/environments/development.rb
Widgets::Application.configure do
  # ...

  # Add this line: disables mutex around each request
  config.middleware.delete ::Rack::Lock

  # Change from false to true!
  config.eager_load = true
end
```

Next, make sure to use a thread-safe web application server like puma; avoid process-based servers like unicorn which can only handle as many concurrent requests as there are processes. To install puma, simply add it to `Gemfile` and run `bundle exec puma -p 3000` instead of `rails server`:

```
# Gemfile
gem 'puma'
```

Now, consider a controller with `ActionController::Live` enabled. It will send server-sent events every second, incrementing a counter each time.

```
# app/controllers/timer_controller.rb
class TimerController < ApplicationController
  include ActionController::Live

  def tick
    response.headers["Content-Type"] = "text/event-stream"

    begin
      seconds = 0
      loop do
        response.stream.write("data: #{seconds}\n\n")

        seconds += 1
        sleep 1
      end
    rescue IOError
      # client disconnected
    ensure
      response.stream.close # cleanup
    end
  end
end
```

As shown in the `TimerController` example above, live streaming is enabled by including the `ActionController::Live` module. When enabled, the controller action can write directly to `response.stream`.

The consumer of the timer API can simply be a static page in `public/`:

```
<!DOCTYPE html>
<html>
  <!-- public/timer.html -->
  <head>
    <title>Timer Example</title>

    <script type="text/javascript" src="http://code.jquery.com/jquery.js"></script>
    <script type="text/javascript">
      $(function() {
        var timerSource = new EventSource("/tick");
        timerSource.onmessage = function(e) {
          $("#timer").text(e.data);
        };
      });
    </script>
  </head>
  <body>
    Timer: <span id="timer"></span> seconds.
  </body>
</html>
```

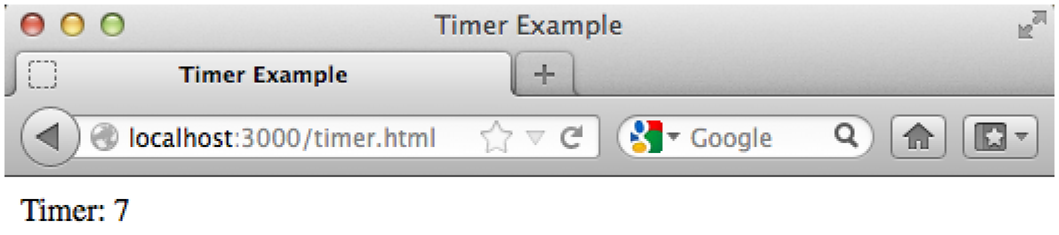
Any time a server-sent event is received, the `#timer` text is updated.

Finally, wire everything up by adding a route for the `tick` controller action:

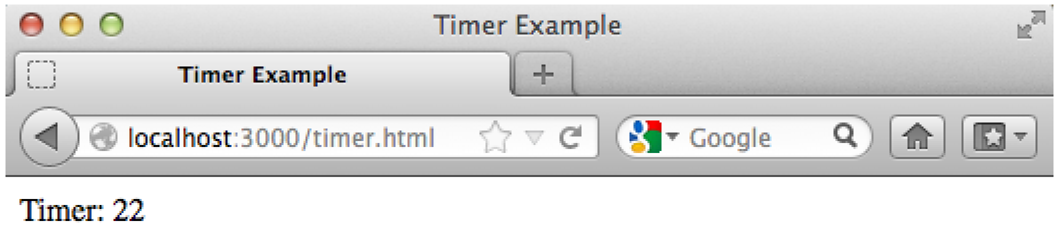
```
# config/routes.rb
Widgets::Application.routes.draw do
  # ...

  get "/tick" => "timer#tick"
end
```

Start or restart puma (`bundle exec puma -p 3000`), and navigate to http://localhost:3000/. The timer should start counting up every second:



Timer: 7



Timer: 22

Voila! Consider using `ActionController::Live` to implement desktop-like applications that react rapidly (avoiding AJAX polling, for instance), or to stream JSON to users on connections with lower bandwidth or higher latency (e.g., cellular data).

Cache Digests

Nesting fragment caches--often called *russian doll caching*--are an effective way to achieve a speed up while keeping view code relatively simple.

Consider an application that manages shopping wishlists: a customer creates a wishlist with a title and adds items he or she wishes to buy to it.

The application might contain a view that renders the entire wishlist and caches the fragment of HTML:

```
<%# app/views/wishlists/show.html.erb %>
<%- cache ["v1", @wishlist] do %>
  <h1><%= @wishlist.title %></h1>

  <ul>
    <%= render @wishlist.items %>
  </ul>
<%- end %>
```

`"v1"` is used to allow developers to easily *bust* the cache for every wishlist by incrementing the string to `"v2"` (and later to "v3" and so on) any time the markup inside the `cache` block is changed. If the `"v1"` were not present or not incremented, the application might display a mishmash of cached content with the older markup alongside updated content with newer markup.

For instance, a designer adding a CSS class to the wishlist title header tag would also need to increment the string to make sure every wishlist title is rerendered with the class after the new application code was deployed:

```
<%# app/views/wishlists/show.html.erb %>
<%- cache ["v2", @wishlist] do %>
  <h1 class="wishlist-title"><%= @wishlist.title %></h1>

  <ul>
    <%= render @wishlist.items %>
  </ul>
<%- end %>
```

Next, consider a partial that renders each item. It might look very similar to the wishlist view:

```
<%# app/views/wishlists/_item.html.erb %>
<%- cache ["v1", @item] do %>
  <li><%= @item.name %> (<%= @item.price %>)</li>
<%- end %>
```

This strategy is called russian doll caching because of its use of nested templates, each of which is cached. The wishlist is an outer "doll," and each item is an inner doll. Caching occurs in both places.

Russian doll caching works well when the outer cache is automatically busted by Rails when a record is updated. For instance, if a wishlist's title is changed, the outer wishlist cache is rerendered on the next request. However, each inner item cache remains intact; therefore, the cached content for each item can be used as the wishlist rerenders, speeding up the process.

As shown, however, russian doll caching breaks down a bit when the markup in the *item* partial (the inner "doll") is changed. In that case, a developer must manually walk up the graph of dolls, incrementing `"v1"` along the way.

In this example, changing the markup in the `_item.html.erb` partial requires bumping `"v1"` to `"v2"` in that file, but *also in `wishlists/show.html.erb`* (the view that renders the entire wishlist). Otherwise, the outer wishlist cache will continue to display the old item markup.

The **cache_digests** plugin that is included with Rails 4 solves the problem.

In Rails 4, the wishlist and item views can be written as:

```
<%# app/views/wishlists/show.html.erb %>
<%- cache @wishlist do %>
  <h1 class="wishlist-title"><%= @wishlist.title %></h1>

  <ul>
    <%= render @wishlist.items %>
  </ul>
<%- end %>
```

```
<%# app/views/wishlists/_item.html.erb %>
<%- cache @item do %>
  <li><%= @item.name %> (<%= @item.price %>)</li>
<%- end %>
```

With **cache_digests**, any time the view where the `cache` call resides is modified, the cache is automatically busted, *including caches in dependent views*. There is no longer a need for `"v1"`.

cache_digests solves the problem we had with wishlists and items: changing the markup in `_item.html.erb` automatically busts the cache there, as well as in `wishlists/show.html.erb`.

cache_digests operates by embedding an MD5 hash of the template content in the cache's key, including template content for dependent views. When the template content changes, the hash value changes. Read more about **cache_digests** via its README on GitHub.

Like certain other Rails 4 features, **cache_digests** can be included as a gem in Rails 3.2, so it can be used today by simply adding the gem to `Gemfile`. I recommend using **cache_digests** in existing applications if you already use this style of caching or want to use it before Rails 4 is released.

Encrypted Cookies

Rails 4 introduces encrypted cookies, and uses them by default (in newly generated apps) as the store for session data. Encrypted cookies save data in a form that cannot be easily tampered with by users; furthermore, users cannot even read the data that is being saved at all.

In contrast, Rails 3 uses digitally signed cookies as the default store for sessions. Digitally signed cookies cannot be easily tampered with, but users *can* read the data that is being saved.

If you do nothing upon upgrading, Rails 4 will continue using digitally signed cookies. However, you can seamlessly transition to the encrypted cookie store to garner an increase in security.

First, generate a new secret with the `rake secret` command:

```
$ bin/rake secret
35137db8a7a7ac525c...
```

Copy that value and open `config/initializers/secret_token.rb`:

```
# config/initializers/secret_token.rb
Widgets::Application.config.secret_token = 'b01f7a8...'
```

There should already be a value for `secret_token`: leave it as-is, but add a new line for `secret_key_base` using the value generated by `rake secret`:

```
# config/initializers/secret_token.rb
Widgets::Application.config.secret_token = 'b01f7a8...'
Widgets::Application.config.secret_key_base = '35137db8a...'
```

With the changes to `config/initializers/session_store.rb` and `config/initializers/secret_token.rb`, the application has been updated to use the new encrypted session store!

Be sure to make this change only after your Rails 4 application is running smoothly in production. While digitally signed cookies generated by Rails 3 are transparently upgradable to encrypted Rails 4 cookies, Rails 4 encrypted cookies will not work correctly with Rails 3. If you end up rolling back your application due to problems with Rails 4, any users whose cookies were upgraded will have their sessions destroyed.

Declarative ETags

Client-side caching is one of the best ways to save sending bytes over the wire, thereby reducing bandwidth costs and load times.

An entity tag (ETag) is one mechanism defined in the HTTP specification that supports caching. When a browser first requests a resource, the server may send back an ETag. An ETag is an opaque hash representing a version of the resource. When a browser requests the same resource in the future, it sends along the ETag it remembered the last time the resource was requested. If the ETag matches the current version of the resource (i.e., the resource has not changed since the last time it was requested), the server can send a "304 Not Modified" response with an empty body, saving bandwidth and server processing time.

Conversely, if the resource has changed, the ETags will not match and the server knows it must send a full response.

An ETag can be generated easily from an ActiveRecord model. By default, the model's class, `id` (primary key), and `updated_at` are combined as the basis for the ETag hash. With `updated_at` in the mix, this `cache_key` returns a new value every time the record is updated.

Rails has provided facilities to generate ETags since Rails 3. Consider a controller that takes advantage of ETags:

```ruby
# app/controllers/widgets_controller.rb
class WidgetsController < ApplicationController
  def show
    @widget = Widget.find(params[:id])
    fresh_when(@widget)
  end
end
```

The fresh_when method sets up the response to include an ETag. Furthermore, if the ETag sent by the browser matches the current ETag, the controller will not render the view, opting instead for a "304 Not Modified" empty response.

Sometimes it makes sense to use additional information when generating the ETag, for instance, to scope the ETag to the currently logged in user. Otherwise, cached content may be reused for a different user (an information leak and security vulnerability!).

In Rails 3, this scoping can be achieved by passing an array of values to `fresh_when`:

```
# app/controllers/widgets_controller.rb
class WidgetsController < ApplicationController
  def show
    @widget = Widget.find(params[:id])
    fresh_when([@widget, current_user.id])
  end
end
```

In this case, the `current_user`'s ID is included when generating the ETag. As a result, different ETags will be generated for every user, even for the same version of `@widget`. Information leak patched!

However, it can be a pain to remember to add this scoping in every controller action. Rails 4 introduces a new declarative syntax for these controller-wide ETag concerns:

```
# app/controllers/widgets_controller.rb
class WidgetsController < ApplicationController
  etag { current_user.id }

  def show
    @widget = Widget.find(params[:id])
    fresh_when(@widget)
  end
end
```

The code above has the same behavior as the previous example where `current_user.id` was passed explicitly to `fresh_when`. However, it is more readable and DRY (Don't Repeat Yourself: an adjective for code or knowledge that is not duplicated).

Furthermore, you can call `etag` multiple times to continue adding values used when generating ETags for the controller.

Use in Rails 3.2

While declarative ETags will be baked into Rails 4 itself, they can also be used in Rails 3.2 applications by pulling in the `etagger` gem:

```
# Gemfile
gem 'etagger'
```

Chapter 18

Turbolinks

New Rails 4 applications include the **turbolinks** gem, which can sometimes make your application faster by avoiding full page refreshes as a user navigates through an application by clicking links.

When **turbolinks** is enabled and a user clicks on a link, the request is sent in the background using AJAX. The response still contains a fully rendered HTML page, so **turbolinks** does not necessarily save much bandwidth. However, it does achieve a significant speedup by splicing out both the `<title>` of the new page and the content in the `<body>` section, and replacing that content in the current document.

Because the page did not refresh completely, the stylesheets (CSS) and JavaScript assets do not have to be parsed again by the browser. Because modern web applications tend to have a lot of CSS and JavaScript, **turbolinks** can make your application feel a lot snappier.

turbolinks is supported in Safari 6.0+ (but *not* Safari 5), IE10, and recent versions of Google Chrome and Mozilla Firefox.

turbolinks is similar to pjax, but ideally operates transparently so that no changes are needed beyond enabling it in your application. In practice, of course, there are gotchas that I touch on later in the chapter.

Adding turbolinks to existing applications

turbolinks is packaged as a gem, so existing applications will need to explicitly install it. New Rails 4 applications are generated with **turbolinks** in the generated `Gemfile`.

Notably, the **turbolinks** gem is compatible with Rails 3 so long as the application uses the asset pipeline.

Add **turbolinks** to `Gemfile` and use bundler to install it:

```
# Gemfile
gem 'turbolinks'
```

```
$ bundle install
```

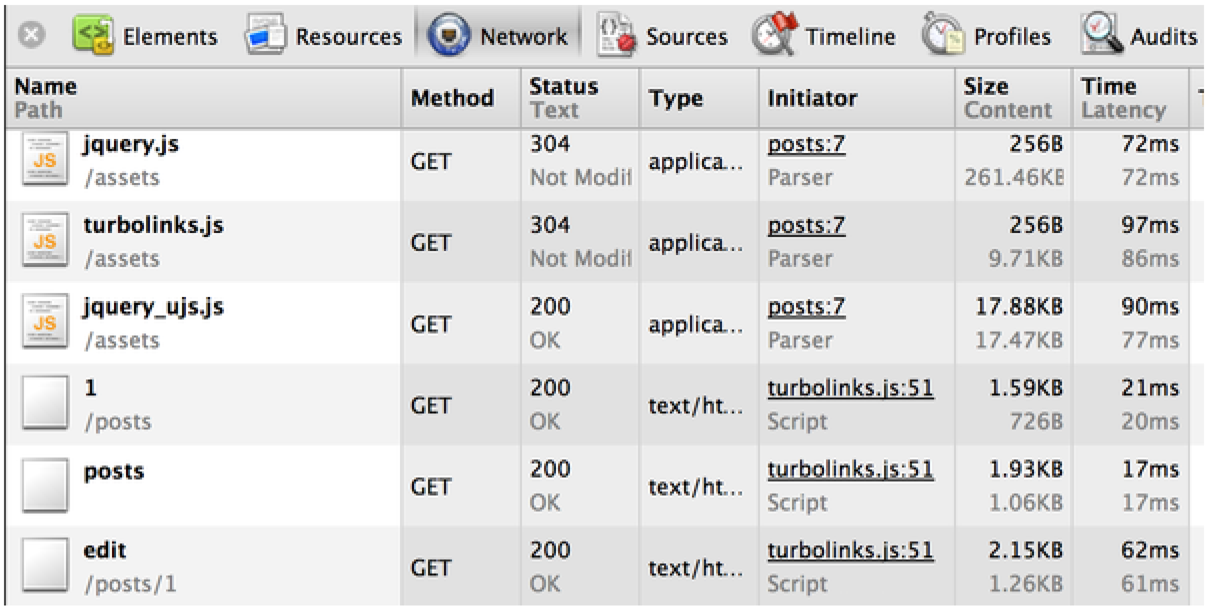
Finally, include `turbolinks` in `app/assets/javascripts/application.js`:

```
// app/assets/javascripts/application.js

// ...
//= require turbolinks
```

After restarting `rails server`, you can verify that **turbolinks** is working correctly by using the web developer tools. In Google Chrome, for instance, developer tools can be enabled via the "Tools ->

Developer Tools" menu item. Navigate to a page in the turbolinkified application and select the "Network" tab:



Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
jquery.js /assets	GET	304 Not Modi	applica...	posts:7 Parser	256B 261.46KE	72ms 72ms
turbolinks.js /assets	GET	304 Not Modi	applica...	posts:7 Parser	256B 9.71KB	97ms 86ms
jquery_ujs.js /assets	GET	200 OK	applica...	posts:7 Parser	17.88KB 17.47KB	90ms 77ms
1 /posts	GET	200 OK	text/ht...	turbolinks.js:51 Script	1.59KB 726B	21ms 20ms
posts	GET	200 OK	text/ht...	turbolinks.js:51 Script	1.93KB 1.06KB	17ms 17ms
edit /posts/1	GET	200 OK	text/ht...	turbolinks.js:51 Script	2.15KB 1.26KB	62ms 61ms

In this example, the last three requests were handled by **turbolinks.js**. You can tell by looking at the "Initiator" column.

From a user's perspective, everything worked as if the entire page *had* refreshed, except it did not; the page was speedier for having avoided loading stylesheets and JavaScript again.

Graceful Degredation

turbolinks gracefully degrades when a browser does not support its features. For instance, in Safari 5 links will simply work as if **turbolinks** were not present at all.

Furthermore, **turbolinks** detects when stylesheets or JavaScript assets have changed on the page being requested. If the assets are different than on the current page, **turbolinks** initiates a full page refresh so that those new assets are correctly loaded.

In that degenerate case, the page will actually be loaded *twice*: one time to check if the assets have changed, and a second time to load in the user's browser.

That degenerate case will perform even worse than if **turbolinks** were not present at all. To avoid this performance nightmare, make sure that the **turbolinks** JavaScript is the last asset loaded in the `<head>` of your application's pages.

In practice this often means making sure that `<%= javascript_include_tag "application" %>` is the *last* piece of JavaScript to be loaded in a layout (the default layout is located in `app/views/layouts/application.html.erb`). If there are additional `javascript_include_tag` or `stylesheet_link_tag` calls after turbolinks, you may suffer performance issues; in that case, consider reordering the `<head>` content so **turbolinks** is last in line.

Gotchas

turbolinks may negatively affect JavaScript that runs code triggered by the jQuery `$(document).ready` event. With **turbolinks** enabled, `$(document).ready` runs only when the first page is ready, not each time a new page is loaded through **turbolinks**.

Consider a view that contains a button:

```
<%# hello.html.erb %>
<button id="hello-button">Hello!</button>
```

Next, consider the corresponding JavaScript code that attaches to the button's `click` event to display an alert when the button is clicked:

```
$(document).ready(function() {
  $("#hello-button").on("click", function() {
    alert("Hello!");
  });
});
```

However, if `hello.html.erb` is displayed through **turbolinks** (i.e., it is not the first page loaded in the web application), the `$(document).ready` event will not be triggered, and the event handler will not be attached to the button.

To fix the problem, it is necessary to attach the event handler both in `$(document).ready` (which will handle the case where the page is the first to load) and `$(document).on("page:load")` (which will handle the case where the page is loaded through **turbolinks**).

```
var attachClickHandler = function() {
  $("#hello-button").on("click", function() {
    alert("Hello!");
  });
};

$(document).ready(attachClickHandler);
$(document).on("page:load", attachClickHandler);
```

The `"page:load"` event is triggered when **turbolinks** loads a new page.

While it is good to know how to manually fix the issue, a library called jquery.turbolinks has also been created to fix the issue described here, ideally without any code changes.

That is, in the presence of **jquery.turbolinks**, the original code will work correctly as `$(document).ready` will be fired in both cases.

Additional Resources

- turbolinks on GitHub
- Railscasts #390: Turbolinks
- Turbolinks Compatibility

Chapter 19

Testing

Directory Structure

Rails 4 has adopted an RSpec-like directory structure by default.

Rails 2 and 3 Directory	Rails 4 Directory	Rails 4 `rake` task
`test/unit`	`test/models`	`rake test:models`
`test/unit/helpers`	`test/helpers`	`rake test:helpers`
`test/functional` (for controllers)	`test/controllers`	`rake test:controllers`
`test/functional` (for mailers)	`test/mailers`	`rake test:mailers`

For backwards compatibility, running the Rails 3 rake tasks (e.g., `rake test:units`) will run tests in both the new directories (`test/models` and `test/helpers`) *and* the old directories (`test/unit` and `test/unit/helpers`). The new rake tasks shown in the table above only run tests in the new directories.

I think this is a great improvement, as the terms "unit" and "functional" were at best opaque and at worst inaccurate.

You should take the proactive step of moving all tests to the new locations after upgrading an application (and before you start writing any new code). You can use the few commands below to move existing tests into the new locations.

```
$ git mv test/unit/helpers test/helpers
$ git mv test/unit test/models

$ mkdir test/mailers
$ git mv test/functional/*mailer_test.rb test/mailers

$ git mv test/functional test/controllers
```

If an application has mailers that are not postfixed with `Mailer` (e.g., `Notifier` instead of `NotificationsMailer`), you will need to move them one-by-one from `test/controllers` into `test/mailers`:

```
$ git mv test/controllers/notifier_test.rb test/mailers
```

Finally, commit the result:

```
$ git commit -m 'Moved tests to Rails 4 conventional locations'
```

Chapter 20

Common Upgrading Scenarios

This section covers common error messages and other behavior that is likely to occur when upgrading applications from Rails 3 to Rails 4.

```
`attr_accessible` is extracted out of Rails into a gem.
Please use new recommended protection model for params
(strong_parameters) or add `protected_attributes` to your
Gemfile to use old one.
```

Rails 4 extracted `attr_accessible` and `attr_protected` into the `protected_attributes` gem. To fix the error, add the `protected_attributes` gem to your `Gemfile`.

Read more about it in the `strong_parameters` section.

```
... was passed as :conditions but is not callable.
Pass a callable instead: `conditions: -> { where(approved: true) }`
```

Rails 4 pushes you to use callable objects when passing conditions to `validates_uniqueness_of`. More information is available in the eager-evaluated scopes section.

```
DEPRECATION WARNING: The following options in your Post.has_many
:recent_comments declaration are deprecated: :conditions. Please use a scope
block instead.
```

Rails 4 deprecates many options to `has_many`, `has_one`, and `belongs_to`. Instead of using `:order` and `:conditions`, for instance, you pass a scope wrapped in a lambda. More information is available in the eager-evaluated scopes section.

```
DEPRECATION WARNING: config.whiny_nils option is deprecated
and no longer works.
```

Rails 4 removed the whiny_nils feature. Read more about it in the ActiveRecord chapter.

To solve the deprecation warning, simply remove any lines that set `config.whiny_nils`. Rails 3 added the configuration by default in `config/environments/development.rb` and `config/environments/test.rb` by default.

```
config.eager_load is set to nil. Please update your
config/environments/*.rb files accordingly:
```

Rails 4 introduces new settings for thread safety. This deprecation warning is prompting you to set the value of the new `config.eager_load` setting. Read more about it in the Thread Safety chapter.

```
DEPRECATION WARNING: Active Record Observers has been extracted out of Rails
into a gem.  Please use callbacks or add `rails-observers` to your Gemfile to
use observers.
```

Rails 4 extracts observers to a gem. Read more about it in the Observers section.

```
DEPRECATION WARNING: The Active Record auto explain feature has been removed.

To disable this message remove the `active_record.auto_explain_threshold_in_seconds`
option from the `config/environments/*.rb` config file.
```

Rails 4 removes Auto-EXPLAIN for inefficient queries. Read more about it in the Auto-EXPLAIN queries section.

```
The provided regular expression is using multiline anchors (^ or $), which may
present a security risk. Did you mean to use \A and \z, or forgot to add the
:multiline => true option? (ArgumentError)
```

Rails 4 does not allow the `^` and `$` anchors with a `validates_format_of` validation. Read more about it in the validates_format_of with `^` and `$` section.

```
You don't have dalli installed in your application. Please add it to your
Gemfile and run bundle install

`rescue in lookup_store': Could not find cache store adapter for
mem_cache_store (cannot load such file -- dalli) (RuntimeError)
```

Your application caches data in memcached. Rails 4 no longer uses the `memcache-client` gem as it has been superceded by the `dalli` gem. Remove the `memcache-client` gem from Gemfile and add `dalli` instead. Read more about it in the caching with memcached section.

Chapter 21

Forking Gems to Loosen Version Constraints

If in the course of upgrading an application to Rails 4, `bundler` detects a conflict between a gem and `rails` (or one of its supporting gems like `activerecord`), it is *possible* that the gem is actually compatible and simply needs its dependency constraints loosened.

If you run into a gem that is not yet compatible with Rails 4, running `bundle` commands will result in an error message similar to:

```
Bundler could not find compatible versions for gem "actionpack":
  In Gemfile:
    simple_form (~> 2.0.4) ruby depends on
      actionpack (~> 3.0) ruby

    rails (>= 0) ruby depends on
      actionpack (4.0.0.beta)
```

The examples in this chapter use the simple_form gem, a gem that makes producing forms easier. However, you can attempt this process for any gem that is currently constrainted to Rails 3 only.

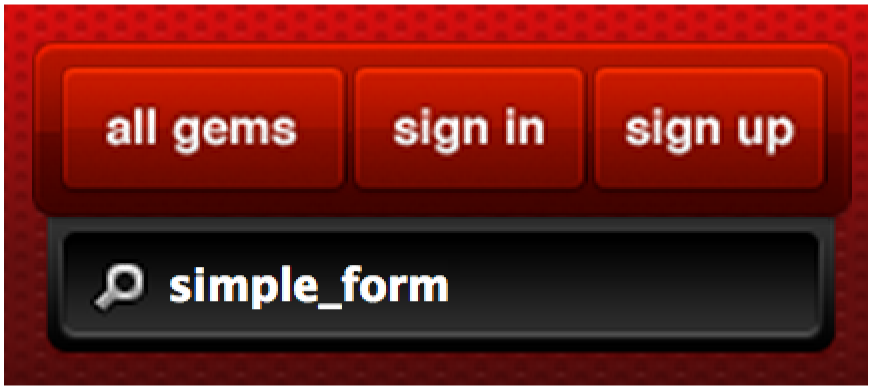
Only use this process if the gem's authors have not yet released a version that is compatible with Rails 4. If the gem turns out to be fundamentally incompatible with Rails 4, this process will unfortunately not help.

Forking the Source

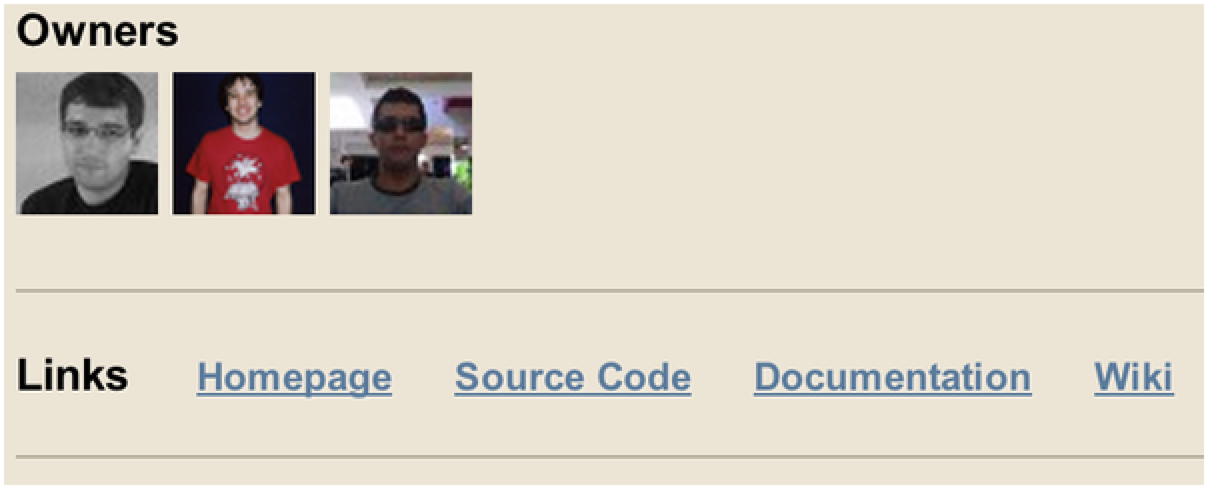
Most Ruby gems' source code is hosted on GitHub. GitHub makes it easy to create a copy of the source code repository for a project in order to modify it. GitHub calls this process *forking*.

To fork a repository, you will need a GitHub account. A free GitHub account is easy to create via sign up page.

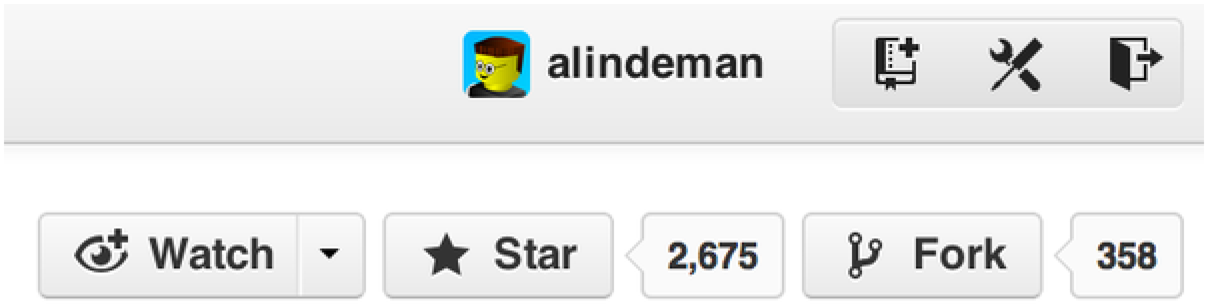
First, find the source code repository for a gem by searching for it on rubygems.org:



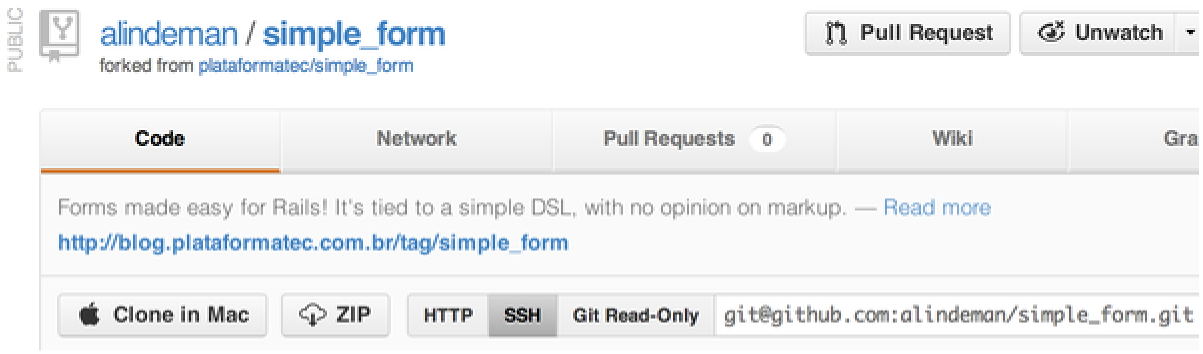
Click on the first result and find the "Source Code" link:



You will likely be directed to the source code on GitHub. Click the "Fork" button in the top right. If there is no "Fork" button, make sure to login to GitHub first.



After a few seconds, the fork will be created:



Forms made easy for Rails! It's tied to a simple DSL, with no opinion on markup. — Read more
http://blog.plataformatec.com.br/tag/simple_form

```
git@github.com:alindeman/simple_form.git
```

Using the SSH or HTTP URL, clone the repository to your local machine and navigate into the directory for the gem:

```
$ git clone git@github.com:alindeman/simple_form.git
$ cd simple_form
```

Look for a file with the extension `.gemspec` corresponding to the gem name. In the case of `simple_form`, there is a `simple_form.gemspec`. Open it in your favorite text editor.

Find lines that call the `add_dependency` method for the `rails` gem or any subproject of Rails such as `activerecord`, `activemodel`, or `actionpack`. `simple_form` currently depends on two such gems:

```
# simple_form.gemspec
Gem::Specification.new do |s|
  # ... other directives ...

  s.add_dependency('activemodel', '~> 3.0')
  s.add_dependency('actionpack', '~> 3.0')
end
```

The '~> 3.0' constraint allows the last digit of the version to vary. For instance, '~> 3.0' allows `activemodel` and `actionpack` version 3.0.x, 3.1.x, and 3.2.x, but not 4.0.x. This is our problem!

Change the `simple_form.gemspec` so it is more forgiving. The easiest way is to replace `~>` with `>=`. `>=` specifies that any version greater than or equal to 3.0 is allowed, including 4.0.

```
Gem::Specification.new do |s|
  # ... other directives ...

  s.add_dependency('activemodel', '>= 3.0', '< 5')
  s.add_dependency('actionpack', '>= 3.0', '< 5')
end
```

Save the file, commit it via git, and push it back to GitHub:

```
$ git add simple_form.gemspec
$ git commit -m 'Loosens constraints to support Rails 4.0'
$ git push origin master
```

Next, navigate back to the Rails application that depends on `simple_form` (or whichever gem you're upgrading) and open up its `Gemfile`. Replace the line that currently specifies `simple_form` as a dependency with a line that points to the newly forked version.

```
# Gemfile

# Replaces gem 'simple_form', 'X.Y.Z'
# Use your GitHub username instead of 'alindeman'
gem 'simple_form', github: 'alindeman/simple_form'
```

Finally, ask Bundler to install the newly updated version of `simple_form` by running `bundle install`:

```
$ bundle install
```

If all went well, output similar to below will be shown:

```
...
Using simple_form (2.1.0.dev) from git://github.com/alindeman/simple_form.git (at master)
...
```

This means that Bundler is using your version of `simple_form` instead of the official one.

It may be necessary to perform these steps for many gems in order to upgrade an application. Again, it may feel a bit like the whack-a-mole game.

If after resolving all versioning conflicts, the newly upgraded Rails application does not run properly, it may be because a gem does not support Rails 4 at a more fundamental level. In that case, file a bug on the GitHub issue tracker for that gem. Make sure to include any error messages or exception backtraces so that the gem maintainers can debug the issue more easily.

Extracted Gems

During the development of Rails 4, many features that were present in earlier versions of Rails were removed from Rails itself and extracted to gems.

During the upgrade process, I recommend pulling in all of these gems to make the upgrade process as smooth as possible. You may not know upfront if your application makes use of one of these features, and it's one more thing to worry about breaking.

After your application has been successfully upgraded, though, consider spending time removing gems that provide features your application does not use.

The table below describes each of the extracted gems and what it provides:

Gem	Description
protected_attributes	Provides **attr_accessible** and **attr_protected** for mass-assignment protection. I expect that most applications will keep this gem, unless you spend time to switch to **strong_parameters**.
activeresource	Provides an ActiveRecord-like abstraction over a RESTful API. You can quickly figure out if your application uses ActiveResource by searching for classes that inherit from `ActiveResource::Base`. More details about ActiveResource are available in the ActiveResource chapter.
actionpack-action_caching actionpack-page_caching	Rails 4 includes many improvements to fragment caching, but action and page caching have been extracted. If your application uses the `caches_page` or `caches_action` directives in any controller, you will need these gems. Otherwise, you can remove them.
activerecord-session_store	The ability to store session data in a database table has been extracted in Rails 4. You need this gem if you store session data in the database by setting `config.session_store :active_record_store` in `config/initializers/session_store.rb`. Otherwise, you do not need this gem.
rails-observers	Rails no longer encourages the use of observers, separate objects that can react to lifecycle events of ActiveRecord models. If you use observers--classes that inherit from `ActiveRecord::Observer`, you need this gem; otherwise, you can remove it. More information is available in the Observers chapter.
actionpack-xml_parser	Following security vulnerabilities involving inbound XML, Rails extracts the ability to accept XML input to a gem. If your application accepts XML in a request body (note: this is distinct from *responding with* XML), you will need to pull in this gem. Otherwise, it can be removed.
rails-perftest	Rails 4 extracts `ActionDispatch::PerformanceTest`. If your application includes performance tests (usually located in `test/performance`), you need to bundle this gem. Otherwise, you can remove it. More information is available in the Performance Tests chapter.
actionview-encoded_mail_to	Rails previously included a little-known feature to obfuscate email addresses in hyperlinks, either with HTML entities or JavaScript code. If your application uses the `encode` option with `mail_to` in any views, keep this gem. Otherwise, it can be

Gem	Description
	removed. More information is available in the activerecord-encoded_mail_to chapter.

Upgrading to Rails 4
Andy Lindeman