Why You'll Love Ruby on Rails

An Examination for ColdFusion Developers

Hal Helms

Foreword

In 2009, I was tricked into learning Ruby on Rails. By that point, I had been a ColdFusion developer for over a dozen years, had written three books on ColdFusion, and had trained hundreds of programmers on developing web applications with ColdFusion. I was an Adobe Community Expert and had been a major contributor to two of the most popular ColdFusion frameworks (Fusebox and Mach-II). Although I was a Sun-certified Java programmer, I preferred to use ColdFusion for the great majority of my work. I was productive and content.

Now, I have a good friend, John Quarto-vonTivadar. John is something of a guru in the world of marketing. He's written a best-selling book on marketing, speaks at conferences, and is the inventor of Persuasion Architecture. He's also a rocket scientist (no exaggeration) and for years was an avid ColdFusion fan.

John had taken up Rails some three years before and I was subject to his wild enthusiasm over this new language/framework combination. Actually, I was more *annoyed* by his gushing over how cool Rails was. At that time, there was a well-publicized video that showed building a blog in something like 7 minutes with Rails. At his insistence, I watched it and reported back to him: "Great. The next time I want to build a blog in seven minutes, I'll definitely use Rails. Until then, will you quit bugging me about it?"

Not only did he *not* quit bugging me, he sneakily appealed to our friendship to provide a "*little* help" with an application he was building using Rails, thereby ensuring that I would have to learn at least a *little* about Ruby and Rails.

I *didn't* like Rails. I fought it. I swore at it. (I may even have impugned its maternal lineage.) John was patient, assuring me that once we were through with the troublesome section, I could go back to my beloved ColdFusion.

While I didn't care for Rails, I did like *Ruby*. I started my programming career using Smalltalk, a language in which *everything* was an object. And here was Ruby with exactly the same thing—everything is an object. That feature means the language can be *very* consistent. That's one of the things I value most in a language: do something one way or another, but stick with it so that developers don't have to constantly turn to the docs to figure out how to do that something.

John knew this, of course. And he knew that, once my interest was piqued, he could count on me wanting to know more. So, while *I* was surprised to find myself signing on to help complete the application, John smiled knowingly.

By the time we finished the application, I was hooked. I was drawn in by the *insane productivity* of Rails, by the *beauty and consistency of the Ruby language*, and by the *enormous number of third-party plugins* (*gems* in Rails-speak) that enabled seemingly

any functionality I needed with just a few lines of code. What in the world had happened? When I began, I actively *disliked* Rails. Now, I was becoming (gulp) a Rails fanboy? Oh, the humanity...

The more I worked with Rails, the more I realized that the thing I loved about ColdFusion was exactly what I was finding in Ruby on Rails: both have a mission of *enabling* developers rather than *enslaving* them with rules. ColdFusion was first. With Rails, it now has some welcome company.

Today, about half my development is in Ruby on Rails; about half is in my old friend, ColdFusion. For certain things, Rails is ideal; for other kinds of projects, ColdFusion shines. I've found the experience of learning Ruby on Rails (and the exposure to new ways of thinking about development) has made me a better, more *balanced* developer whichever technology I choose to work in.

In this short guide to Ruby on Rails for ColdFusion developers, I share why I believe, if you love ColdFusion, you'll love Ruby on Rails as well.

Why You'll Love Ruby on Rails

I. It's Insanely Productive
II. It Has Third-Party Plugins for *Everything*
III. It's Open Source
IV. It's in Huge Demand in the Workplace
V. It's Easy to Learn

I. It's Insanely Productive

Insanely productive? Really? Every few years, a new technology rolls through the IT world promising outrageous benefits. It doesn't take long before one learns to *heavily* discount all the breathless claims issued by "analysts" who seem to make a living solely from touting the Next Big Thing.

Instead of relying on them, let's listen to seasoned developers for their take on Ruby on Rails. Most readers will be familiar with Martin Fowler, whose books[1] bring deep insight to software engineering. Here's Fowler's take on Rails:

"*It is impossible not to notice Ruby on Rails. It has a huge effect both in and outside the Ruby community...Rails has become a standard to which even well-established tools are comparing themselves.*"

That's a nice quote—but a long way from establishing an *insane* standard of productivity. For something much closer to that, listen to James Duncan Davidson. Davidson is the creator of both Tomcat[2] and Ant[3].

“Rails is the most well thought-out web development framework I’ve ever used. And that’s in a decade of doing web applications for a living. I’ve built my own frameworks, helped develop the Servlet API, and have created more than a few web servers from scratch. Nobody has done it like this before.”

That's pretty high praise from a pretty accomplished programmer. But I'm getting ahead of myself; I still haven't told you *what* Rails is.

Ruby on Rails is a full-stack open source web application framework built on top of the Ruby language. The best analog in the ColdFusion world would be the combination of ColdFusion with one of the popular, powerful frameworks such as Mach-II, ColdBox, or Model Glue. And here are the key features of Ruby on Rails:

- built on the Model-View-Controller design pattern
- provides a built-in ORM using the ActiveRecord design pattern
- automatically builds stub unit and integration tests

---

[1] *Patterns of Enterprise Application Architecture, Domain Specific Languages, Analysis Patterns* among others

[2] Tomcat is an extremely popular open source Java servlet container.

[3] Ant is a widely-used program for automating software build processes.

- automatically provides for independent environments for development, testing, and production
- offers powerful, collaborative database administration through *migrations*
- provides a task-automation language called *rake*
- creates standardized file and package system
- manages third-party additions through a provided utility, *Bundler*
- creates customizable code for all CRUD functions
- supports RESTful and SOAP web services

So, that's *what* Ruby on Rails is. But the claim is: *Rails is insanely productive*. How so?

Rails does this by *analyzing* every piece involved in web development, then *transforming* each piece, making it as *simple* and flexible as possible.

Example: Code organization has bogged down more than one development project. Anyone who's been brought onto a project already in the works has experienced this. We may need to change the code to update the shopping cart, say, but among hundreds or thousands of files, which file is it that needs to be changed—and where is it? Rails solves this by automatically creating a complete directory structure for every possible type of file.

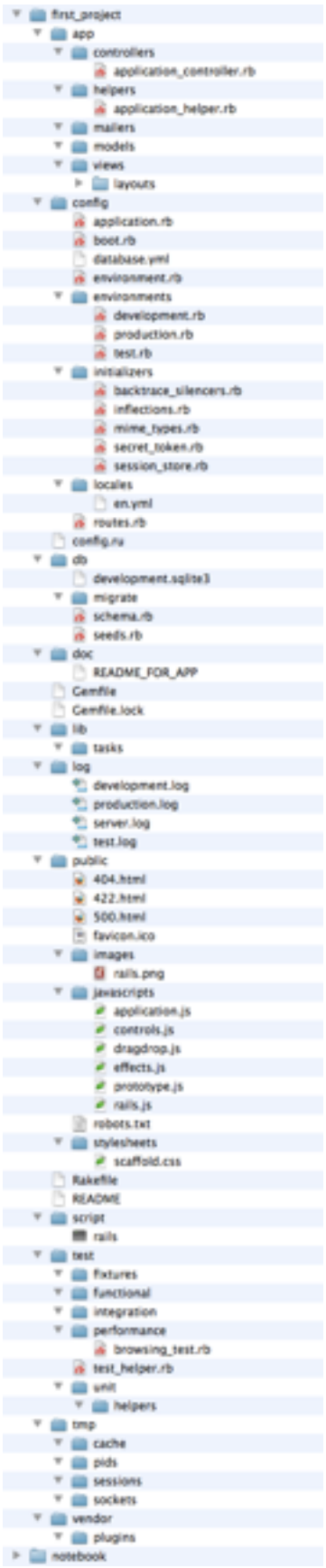
first_project
app
controllers
application_controller.rb
helpers
application_helper.rb
mailers
models
views
layouts
config
application.rb
boot.rb
database.yml
environment.rb
environments
development.rb
production.rb
test.rb
initializers
backtrace_silencers.rb
inflections.rb
mime_types.rb
secret_token.rb
session_store.rb
locales
en.yml
routes.rb
config.ru
db
development.sqlite3
migrate
schema.rb
seeds.rb
doc
README_FOR_APP
Gemfile
Gemfile.lock
lib
tasks
log
development.log
production.log
server.log
test.log
public
404.html
422.html
500.html
favicon.ico
images
rails.png
javascripts
application.js
controls.js
dragdrop.js
effects.js
prototype.js
rails.js
robots.txt
stylesheets
scaffold.css
Rakefile
README
script
rails
test
fixtures
functional
integration
performance
browsing_test.rb
test_helper.rb
unit
helpers
tmp
cache
pids
sessions
sockets
vendor
plugins
notebook

Figure 1: Directories and files generated by the command: **rails new first_project**

From this example, you can see why Rails is called *opinionated software*. It makes certain assumptions. Rails speaks of *convention over configuration*. By accepting Rails' defaults, you're free of the tedium of configuration. Don't like some of the defaults? You can always fall back to configuration to change them.

When I first started, I wanted to override *all* the conventions. An experienced Rails programmer suggested that, while I was learning, I do things the way Rails expected. Later, if I still wanted to change things, I could always do so. As I became more comfortable with Rails, I realized that accepting the conventions—even when they clashed with my pre-conceived notions—gave me a tremendous boost in productivity. In the clash between ego and productivity, productivity won out.

Example: While we want our work to be exemplary, external pressures can force us to skip some best practices. Rails helps by *baking in* best practices into the framework itself.

- Rails implements the Model-View-Controller design pattern.
-
- Code-coverage unit and functional test stubs are automatically created by Rails.
-
- Separate modes for *development, testing,* and *production* are supported. Each has its own database and different modes can use entirely different databases (*e.g.* SQLite for development and MySQL for testing and production).
-
- Databases can have test data inserted prior to testing and automatically cleaned up after tests are run.

Example: Rails provides a clean, flexible way to administer databases through *migrations*. Changes to the database structure are stored in separate files. Rails keeps track of which changes have been implemented and changes can always be rolled back, providing a very useful versioning construct for managing your database.

Example: Before accepting input from sources outside our control (web forms, for example), we can use Rails' built-in dead-simple validations. This goes beyond trivial "Is this an email?" validation. Look at the code that ensures a new **User** has selected a **username** that's unique in our database.

```
class User < ActiveRecord::Base
  validates_uniqueness_of :username
end
```

That's all that's required. Any attempt to create a new **User** must first pass the uniqueness validation check. Rails validation makes very sophisticated validation checking a simple one-line exercise.

Example: Many web applications need basic functionality for creating, editing, viewing, listing, and deleting application-specific classes—**Users**, **Customers**, and **Products**, for example. Basic CRUD functionality for, say, a **Customer** (complete with forms for creating/editing), we can make use of a command-line program called rake. Now a single command will generate an entire mini CRUD application: `rails g scaffold Customer first_name:string, last_name:string, credit_limit:integer`.

With that, a class model for **Customer** is created as well as a **Customer** controller (with all CRUD actions accounted for), and even CRUD views are generated. (While the view code written by these *generators* is seldom exactly what is needed, they can easily be edited to conform to exactly what is required with a minimum of code and effort.)

Example: Web forms are not difficult—but they can be exceedingly tedious. The same form must often serve to both create and edit, forcing ugly (and fragile) conditional code to clutter the form page.

Take, for example, the simple case of the user selecting a **state** field for their address. If the form is in *create* mode, no state will be pre-selected, but if the form is in *edit* mode, we wish to pre-select the state the user had previously chosen. With most frameworks, the developer has to write several lines of conditional code to accommodate this. Rails makes it much easier:

```
<%= select(:address, :state_id, [['Alabama', 1], ['Arkansas', 2], ...]) %>
```

There's no need for any conditional code; Rails will detect if the current address' **state_id** matches one of the options. If so, that option will be selected.

These are just a few examples in a large catalog of Rails productivity features. Although no formal testing (that I'm aware of) has been done to establish the cumulative impact of all these features, Rails developers coming from other languages (including me) are initially struck by just how *much* faster Rails development is.

II. It Has Third-Party Add-Ins for *Everything*

OK, not *everything*, but the list of third-party add-ins (*gems* as they're known in the Rails world) is truly impressive. The number of gems available is already in the thousands -- and continues to grow. Here is just a sampling of the features added to stock Ruby on Rails by various gems:

- automatic database versioning with auditing
- image processing
- automatic testing
- broad variety of ecommerce and payment gateway solutions
- LDAP support
- queuing
- authentication and authorization
- form builders
- pagination
- search
- tagging
- scheduled events
- reporting
- large variety of calendars
- about any kind of code metrics you could wish for
- numerous content management systems
- automatic administrative backend
- continuous integration
- integration with no-SQL data stores
- documentation tools
- email sending and processing
- discussion forums
- game libraries
- geocoding and mapping
- graphing
- HTML parsing
- I18n localization
- CoffeeScript, SASS, and HAML support
- microformats
- mocking frameworks
- music and MIDI libraries
- RSS parsing
- server monitoring
- wikis
- XML processing

Without some way of managing so many gems, the possible dependencies between gems might grow into a snarled mess. The Rails community has forestalled this potential problem by creating a program that manages all gem dependencies.

Gems (and their packaging mechanism) are an elegant solution to adding functionality to a framework, but what if developers are unhappy with *core* components of the framework? The Rails source developers anticipated this, too: core components are highly modularized and can be swapped out for others. Unhappy with the ActiveRecord ORM that ships with Rails? You can use one of several others available, or—the ultimate freedom offered by open source software—you can write your own.

III. It's Open Source

In 1995, when ColdFusion was first released[4], there was one dominant model for releasing software. A company assembled the bits onto a physical medium and released it. Release cycles of 12-24 months were common.

Gradually, a new model has gained favor: *open source*. While commercial products typically restrict access to the source code, open source, as its name implies, provides access to the source code, allowing developers to make changes to it. Of course, most programmers are content to simply *use* open source software. But open source software allows developers to *self-select* to be contributors to the source.

When open source software is successful (much of it never attracts any contributions from any other than its original author), the value provided by this model can be immense. Take, for example, a change in the outside world that has (or should have) an impact on the software -- perhaps something like the advent of Twitter. Marketing tells us "Our customers want integration with Twitter!"

In the proprietary, commercial model, any change (integration with Twitter, in this case) must be evaluated, decided on, scheduled, developed, then put into a release. In the wild-west, open source model, developers can immediately and independently offer fixes to the code that incorporates Twitter integration. Users can choose which of the fixes they prefer. The best changes populate; the others are simply ignored.

Some open source zealots have argued that "software wants to be free" and that the closed/proprietary model is evil. This ignores the fact that a great deal of excellent software is, in fact, proprietary and that it's exactly this proprietary software that forms the theme on which many open source variations are based. Different people can come down on different sides of this argument. What can't be disputed, though, is that the open source model presents a vastly superior *potential* for better software by providing a mechanism for rapid innovation that closed source software does not have.

For that small percentage of open-source software that reaches critical mass, the rapid ready-fire-aim approach that open source projects encourage allow the open-source product's new features to quickly appeal to more users, further encouraging more developers to add their efforts. Compared with this rapid-iteration approach, 12-24 month release cycles are like heavy chains that keep the closed source product developers (who may be excellent) from keeping up with the innovation of the open source model.

---

[4] The same year, incidentally, that Ruby was publicly released. Given some of the some similarities between the two explored later in this paper, I've begun wondering if Rails and ColdFusion were *separated at birth...*

Several years ago, responding to the sea change in web development brought about by the adoption of Ajax, ColdFusion was quick to offer several ColdFusion components that had Ajax "baked in". But the world was changing rapidly and within a few months, jQuery reset the idea of how clients would use Ajax. Now, much of the Spry- and ExtJS-based components needed to be refreshed. But a refresh would have to wait: such is the nature of long product refresh cycles.

Rails, too, faced this situation. Initially, Rails used the Prototype framework. When jQuery went viral, independent developers jumped in, offering trivially easy ways to swap out Prototype for jQuery. Even though Rails itself has lengthy release cycles (like ColdFusion), the open nature of the framework meant that developers could make the changes to the framework needed and deploy those changes in days. Without access to the source code, changes this integral to the framework would have been ugly hacks at best—or impossible at worst.

IV. It's in Huge Demand in the Marketplace

Why one technology experiences explosive growth while another, equally worthy, never achieves wide adoption is a mystery.

The first language I worked with was Smalltalk[5]. Smalltalk was years ahead of its time. Its superiority was so great that we Smalltalkers assumed that, eventually, the world would catch on. But Smalltalk's popularity never ignited and, for better or worse, Java was the language that achieved wide-scale adoption as the first "language of the web"[6].

Today, Ruby on Rails has garnered the accolades and adoption of technologists and employers alike. Although all attempts to accurately gauge a technology's popularity will be imperfect, the demand for Ruby and Rails is clear.

Figure 2: Comparison of job trends from indeed.com

As a recent "BusinessInsider" analysis states, "it's nearly impossible to be an unemployed Ruby on Rails developer".

All technologies have well-defined lifecycles.

---

[5] The creator of Ruby, Yukihiro "Matz" Matsumoto, has stated that one of the influences on Ruby was Smalltalk.

[6] Java, too, was influenced by Smalltalk. In fact, one wag opined that Java was "Smalltalk dressed in C clothing".

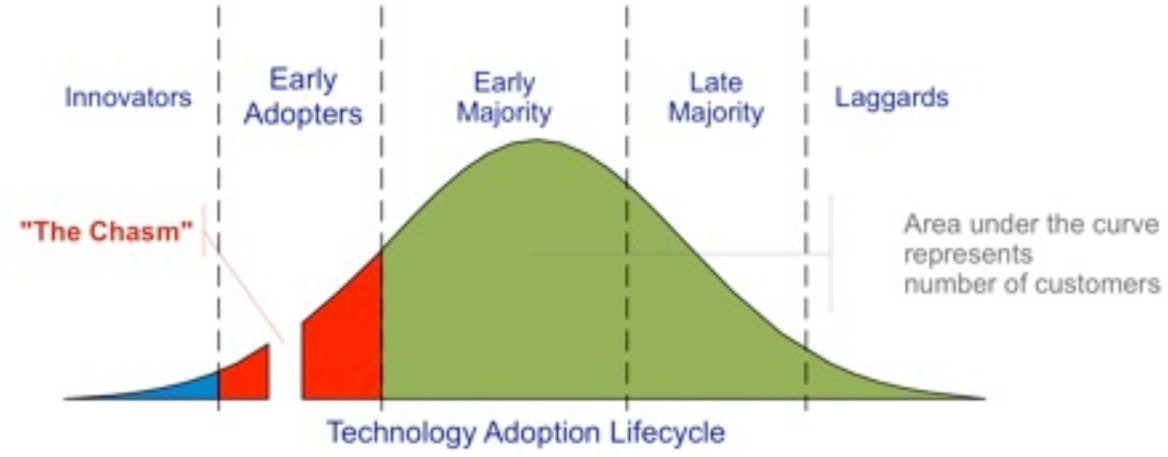

Figure 3. Technology Adoption Lifecycle popularized by Geoffrey Moore in Crossing the Chasm

Widely popular technologies tend to have lengthier cycles, but, at some point, the torch passes to a new technology. But we know that. We know that to be a developer means to be engaged in a lifelong pursuit of knowledge. Having lived (and survived) several technology cycles, I've learned that we're not *ColdFusion* developers or *Ruby on Rails* developers. We're *developers* who learn to be proficient at using multiple technologies.

V. It's Easy to Learn

I'm struck by how much ColdFusion and Ruby on Rails have in common. The biggest similarity, by far, is in their approach to web development: *make commonly done things easy*.

To developers in more rigid languages (*e.g.* Java, C#), Ruby on Rails' approach is a radical change in what they're used to.

No compilation cycle?

No static type checking?

That's crazy talk!

To ColdFusion developers, Ruby on Rails is like the cousin you didn't know you had. It turns out that our cousin is fun to hang out with.

Automatic prevention of cross-site scripting?

Sure!

Mix-in support by including code?

Got it.

Support for calling missing methods in classes?

Why not?

Arrays of heterogeneous data?

Automatic accessors/mutators?

Default method arguments?

Yes, yes, and yes. Hey, we're here to make your life easier, not harder!

ColdFusion has all of these, too, of course. The syntax of Ruby and ColdFusion is different and while there are some unique features that Ruby has, beneath the surface differences lies the same commitment: *empower the developer; make their life easier.*

With so much in common, it's only natural that developers steeped in ColdFusion will feel *comfortable* with Ruby on Rails.

And where there are differences, they're likely to delight you. Some examples:

- *Everything* in Ruby is an object—even numbers. This makes the language extremely *consistent*—learn one way of doing things and it holds true for all classes.

- Ruby is entirely open—both to change and to subclass. Ruby's default implementation of an **Array** allows it to hold a mixture of different data types (like ColdFusion's). Need an array that allows only objects of a certain class—or one that assures the array holds no duplicates? You can subclass **Array** and mold it to meet your needs. (That's true for all classes—including core Ruby and Rails classes.)

- Structs (*hashes* in Ruby-speak) are ordered. You can loop over a hash and things will come out in the same *order* you inserted them.

- Splat arguments are similar to ColdFusion's optional function arguments. In this code, `*language` is the "splat" argument:

  ```
  def splat_test( who, *language, proficiency )
  end
  ```
  [7]

- Ruby recognizes the first argument as **who** and the last argument as **proficiency**. Any number of arguments between the first and the last are automatically put into an array called **language**.

- Variables are private to the block in which they're declared; there's no need to use **var**.

- Ruby goes out of its way to make things as readable as possible. Need to find the date as of 3 days ago? Use this: `3.days.ago`[8]. Want to loop from 1 to 10? This will do it: `for i in 1..10{ }`

- A common practice among programmers in Ruby is the use of **?** on a method end to indicate the method returns a Boolean. Similarly, they use **!** on methods that permanently alter something (deletion, for example). Combined with a conditional, their code often looks like this:

---

[7] Ruby uses **def** and **end** as boundaries on a method instead of C--style curly braces.

[8] Does it look like we're using the number 3 as an object? We are. In Ruby, *everything* is an object—even numbers.

```
product.restock if product.below_restock_level?
```

The goal is to make Ruby read as much like a natural language as possible.

Closing Thoughts

If you're a programmer who loves the simplicity, elegance, and high productivity of writing ColdFusion apps, Ruby on Rails is for you. While keeping all that we love about ColdFusion, Rails introduces new ideas about web development and a new language that is a delight to use.

Does learning Rails mean abandoning ColdFusion? Not at all. Today, two years since I began learning Rails, I find that I use both ColdFusion and Rails. Different projects seem better for one or the other. With the high demand for Rails programmers (be warned: I routinely get 3-4 contacts from headhunters a day), I've found that learning another technology (especially one so similar in spirit to ColdFusion) has made me a better, more in-demand developer.

To all my fellow ColdFusion developers, my advice is this: *try* Ruby on Rails, then let me know your take on Rails by contacting me at hal@halhelms.com. Let me know if you find yourself loving Ruby on Rails!