Examining Ruby on Rails

A Guide for Managers of ColdFusion Development Shops

Hal Helms
hal@halhelms.com

About the Author

I've been an enthusiastic developer and fan of ColdFusion for over 15 years. For years, I wrote a monthly column in "ColdFusion Developer's Journal". I've written three books concerning software development with ColdFusion. I've trained hundreds of ColdFusion developers and have spoken at dozens of ColdFusion conferences. I also developed two of the most popular frameworks for ColdFusion, Fusebox and Mach-II.

I hope you'll find this brief guide to adding Ruby on Rails into your ColdFusion shop helpful. If you have specific questions or would simply like to hear, first-hand, the experiences of a ColdFusion developer, I'd love to hear from you. Write me at hal@halhelms.com.

Hal Helms

Insane Productivity? 3

Productivity—But At What Price? 3

Everything From a Single Vendor? 3

The Personnel Issue 3

Should You Bother Reading Further?

This brief guide is intended for managers responsible for web development.

More specifically, it's intended for ***dissatisfied*** managers responsible for web development.

What's not to like about managing projects for the web?
- projects take too long to complete
- customers aren't thrilled with what's produced
- adding more resources often fails to deliver any benefit
- developers are expensive (and may be hard to find as well)

If you find yourself in any of these categories, you can at least take comfort knowing that you have company. ***Lots*** of company.

The Standish Group, a consulting company that tracks project success rates for custom software, reports that for 2010, more than 66% of projects were "challenged" (as they charitably called those that were more than 200% over budget down to those that were completely abandoned).

Was 2010 an aberration? No. For the seventeen years the aptly-named CHAOS Report has been published, there's been no noticeable improvement—this despite the number of silver bullets that managers were promised would "change everything".

This guide does *not* purport to change everything. It does, though, show how companies faced with vague, changeable requirements have been able to use a **web application framework** called *Ruby on Rails* to take back control of **runaway software projects**.

Who are these companies? Here's a very abbreviated list:
- Twitter
- Hulu
- Groupon
- Yellow Pages
- Bayer Advanced
- Get Satisfaction
- ChowHound
- FunnyOrDie
- 37 Signals
- Zendesk
- Odeo
- 43 Things

These are different companies with differing business models in different industries. What is it about Rails that makes it the **choice of so many successful companies**? For the answer to that, read on.

Insane Productivity?

Every few years, a new technology rolls through the IT world. Analysts and consultants promise fantastic benefits. But after the seminars have been attended, the books sold, the consulting dollars received, too often these **promises** are found to have been **greatly exaggerated**.

Ruby on Rails has received enough hype to make cautious those of us who have watched The Next Big Thing cycle repeat a few times. Analysts and consultants follow the money—not necessarily the truth.

Perhaps our best move is to **ask respected developers** for their take on Rails. **Martin Fowler** is one such developer. His books[1] bring deep insight to software engineering and are prized by other developers. 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."

Then there's this from **James Duncan Davidson** (Davidson is the creator of both Tomcat and Ant[2]):

"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."

What has experienced developers so exited? Ruby on Rails is a full-stack open source web application framework built on top of the Ruby language. Ruby on Rails:

- can run **independently**, on the **JVM** (Java virtual machine), or on the **.NET** platform
- is built on the **Model-View-Controller** (MVC) design pattern
- provides a built-in **database mapping** facility (ORM) using the ActiveRecord design pattern

---

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

[2] Tomcat and Ant are extremely popular Apache Foundation open source tools used by millions of developers.

- automatically builds stub **unit and integration tests**
- 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**

Beyond the bullet points, Rails derives its power by **analyzing every piece** involved in web development. It then **transforms that step**, making it as **simple** and **easy** as possible. The cumulative effect of this cycle of improvement is that developer productivity skyrockets.

Some examples:

- The simple matter of **code organization** has bogged down more than one development project. Each developer has their own preferred way of organization, tending to **chaos**. Rails solves this by prescribing and automatically creating a **standard file and directory structure** for every possible type of file that developers are expected to work within. You can see why Rails is called *opinionated software*: it makes certain assumptions—something Rails calls *convention over configuration*. Accept the conventions and Rails will do a great deal of behind-the-scenes work for you[3].

- A major source of problems is the presence of *bugs* in production software. Experienced developers know that the best way to eliminate bugs is to have **automated testing** in place. A suite of tests protects not only the initial release, but what may be years worth of changes and maintenance to the existing code. But with tight deadlines and limited resources, it's just too easy for developers to skip this step. Rails understands—and so creates **unit and functional tests** stubs. With the drudgery of the testing framework taken care of, developers can write only the tests they need to ensure code integrity.

- To help ensure that code does not turn into a tangled mess, Rails implements the venerable **Model-View-Controller architecture**, a best-practice in web

[3] If the conventions just don't work, Rails allows you to specify your preferences in configuration files.

development.

- Because so much web development code must deal with databases, Rails has particularly **strong support for database interaction**. Developers *describe* their database structure to Rails, which then creates **versioned SQL** files targeted to the specific database. (All popular databases are fully supported.) Rails will then populate the database with sample data for development and testing.

- The best development plans call for **separate environments for *development, testing,* and *production***. Rails does this **automatically**—part of its notion that developers should be freed from drudgery to concentrate on the task of delivering great software.

- Data validation from user forms is another time-consuming chore. It's absolutely necessary, but something that impatient users have little understanding of. Rails provides very **sophisticated validation** that checks data automatically and that is implemented with a **single line of code**.

- On top of application-specific features, many web applications need the same basic functionality for creating, editing, viewing, listing, and deleting. Developers refer to these as CRUD features (for **C**reate, **R**ead, **U**pdate, and **D**elete). While the details differ for different aspects of a program (a *Customer* has different attributes from a *Product*), the code for different components is 90% identical. Within seconds, **Rails *scaffolding* builds all of the code to implement CRUD functions**.

As helpful as the built-in Rails features are, Rails developers benefit from an initial decision by Ruby's creator to offer a standardized feature for **extending** the utility of the **language** through a mechanism known as *Ruby gems*. Gems are used *extensively* in the Rails community to provide features that integrate seamlessly with Rails. This means that, instead of writing new features from scratch, the Rails developer first looks to see if there is already a *gem* available that accomplishes what they need.

These are a few of the features that Rails developers employ to rapidly build web applications. But what of the **costs** of developing in Rails? In the next section, we'll examine the costs of adopting Ruby on Rails.

Productivity—But At What Price?

Productivity is often used as though it consisted of a single vector: **quantity produced**. But productivity is a ratio of **work produced / cost of production**. When we say that Rails is highly productive, that implies something about the costs of developing in Rails.

What can we say about those costs? Ruby on Rails is only five years old—too little time for case studies to have been drawn. We must, therefore carefully look at anecdotal evidence.

When Java- and .NET-based software development house, Relevance LLC, introduced Ruby on Rails into their practice, they documented their results. Writes Stuart Halloway on Relevance's blog:

For the past 18 months, we have been quietly bidding web projects with both Java and Ruby on Rails. The numbers for us, so far, fall out like this:

- ***For applications in the Rails sweet spot (CRUD +AJAX on the web) our Rails price tends to be 30-50% less than the same bid implemented in Java.***
- ***For applications that are nowhere near the Rails sweet spot (do you know what these are?), our Rails price tends to be only 10% less than the same project implemented in Java.***
- ***Applications are completed twice as quickly in Rails.***

That is appealing—but seasoned managers know that far more is spent on ***maintaining* software** than is spent on its initial construction—**5-9 times more** according to multiple studies. These costs are often hidden but their impact on an organization are very real. While all sorts of programs, frameworks, and methodologies exist for *developing* software, very little thought is given to how that same software will be *maintained*.

With built-in features like automated testing, an MVC architecture, and a wealth of third-party plugins, Rails supports not only initial code development but **ongoing code maintenance**.

Everything From a Single Vendor?

ColdFusion is a **powerful language** with **great capabilities**. With so much going for it, though, **ColdFusion has a problem**. In 1995, when ColdFusion was first released, there was one dominant model for releasing software. A company assembled bits onto a physical medium and released it. Release cycles of **18-24 months** were common.

Gradually, a new model has gained favor: ***open source***. The value provided by allowing access to a language's source control can be immense. Under the old, closed source, scheduled-release model, if a new feature was desired, the case for it had to be made to the product manager. It would then be evaluated, approved, scheduled, developed and put into a release.

Contrast this with the wild-west, open source model in which developers can **immediately** and independently **offer fixes** to the code that incorporate the desired change. Users can choose which of the fixes they prefer. The best changes populate; the others are simply ignored.

Compared with this rapid-iteration approach, 18-24 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.

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 ExtJS- and Spry-based components offering Ajax integration. But within a few months, jQuery became the *de facto* choice of libraries for building "Web 2.0" applications. jQuery needed to be integrated with ColdFusion components. 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.

To be **dependent on a single source for innovation** in a fast-paced environment—no matter how excellent the source—necessarily means that we will give up a **competitive edge** to those with access to both the most current technology and the breadth of technology available through the open source model.

The Personnel Issue

ColdFusion managers are not unfamiliar with personnel problems. Senior ColdFusion developers are **harder to find** and **more expensive** than senior developers in other languages. What of Ruby on Rails developers?

One of the driving forces behind the relative expensiveness of ColdFusion developers is their scarcity. The **Gartner Group** estimates that the number of Rails developers is approaching **2 million**. ColdFusion developer estimates is very difficult to obtain, but appears to be between 350 and 700 thousand. That means that right now, for every **1 ColdFusion developer**, there are approximately **4 Rails developers**.

Of course, the demand for Rails developers is also very high. Developers—ever sensitive to technologies needed to remain competitive in the marketplace—are adopting Ruby and Rails at a very fast pace. In fact, **Gartner** estimates that the number of Ruby on Rails developers will **double to 4 million by 2013**.

An article in "Information Age" repeated this theme: "A June [2009] survey by software development research company Evans Data found that Ruby use in North America alone has jumped 40% in the past year, with 14% of developers now using it at least part of the time. In emerging economies, it's even bigger."

But in the "help is coming!" department, there may be an even bigger factor for ColdFusion development managers. There exists a **natural affinity** between **ColdFusion** and **Ruby on Rails** that makes it significantly easier for ColdFusion developers to add this language to their toolset. Similarities between the two include:

- Dynamic type checking
- Mix-in support through includes
- Automatic prevention of cross-site scripting attacks
- Support for calling missing methods
- Arrays hold heterogeneous data types
- Automatic getters and setters
- Default method arguments

Syntax differences are superficial; language philosophies are not. ColdFusion has, for seventeen years, undertaken the lonely task of arguing that languages *should make life easier for the developer, not the machine.* With Ruby on Rails, ColdFusion now gets some welcome company.

Successful Rails Integration

Below are **best practices** discovered by ColdFusion shops for adding Rails to their existing teams.

Find an internal Rails champion

The best approach for introducing Rails to your team is to find a developer enthusiastic to learn new technologies and make that person an advocate for adopting Rails. Their enthusiasm can then spread to the rest of the team while the advocate further serves as a resource for other team members who want to learn.

Start with a small "skunk works" project

An early success can do a great deal to spur adoption. Instead of choosing a large, high-profile application as an initial Rails project, consider a much smaller, "under-the-radar" application. The lack of visibility (and pressure) will give Rails adoption on a real project a much greater chance for success.

Feed your team with learning lunches

A learning lunch is one in which the company brings food in-house for developers. During lunch, your Rails champion gives us 20-30 minute presentation on some specific aspect of Rails. This typically leads to another 20-30 minutes of questions and discussion as team members process new information. As one develop manager put it, "For 10 bucks, I get an hour of a highly engaged developer's time."

Consider some form of mentoring or training

At some point, your internal Rails champion may wish to bring in experts for more in-depth training or mentoring. This is best done at the point at which you have buy-in from your entire team. (You can find high-quality resources here: www.engineyard.com/ruby_on_rails_training.)

Create an online resource list

Another job for your Rails champion: create a list of vetted resources that team members can rely on. Engine Yard, a premiere source of Rails expertise and hosting, can provide a start on such a resource: www.engineyard.com/videos.