A Rails / Django Comparison

Ben Askins and Alan Green

This paper was originally presented at the Open Source Developer's Conference, which ran 5-8 December, 2006 in Melbourne, Australia. It was reviewed at that time, and selected for publication here as being of an excellent standard.

Abstract

Ruby on Rails (“Rails”) is the dominant web programming framework for Ruby and, even outside the Ruby community, is considered the epitome of the latest generation of high-productivity, open source web development tools. Django is one of many competing web development frameworks for Python. It is notable, first, for being highly regarded amongst Python programmers, and second, for being one of the few of the new generation of framework that does not ape Ruby on Rails. Both Rails and Django claim greatly enhanced productivity, compared with more traditional web development frameworks.

In this paper, we compare the two frameworks from the point of view of a developer attempting to choose one of the two frameworks for a new project.

1. Introduction

Ruby on Rails (“Rails”) is the dominant web programming framework for Ruby and, even outside the Ruby community, is considered the epitome of the latest generation of high-productivity, open source web development tools. Django is one of many competing web development frameworks for Python. It is notable, first, for being highly regarded amongst Python programmers, and second, for being one of the few of the new generation of framework that does not ape Ruby on Rails.

This paper is written for developers wanting to choose one of these two frameworks for their own development. For the purposes of this comparison, the authors wrote a specification for a small web application that they implemented twice: once by Ben in Rails, and once by Alan in Django. The implementations were then compared, both quantitatively by code size and implementation time, and with more qualitative measures such as examining the “flavour” of the HTML templating languages.

There are of course other areas that may also be relevant to developers choosing between Rails and Django which are not examined in this paper. This includes relative and absolute performance, amenity to deployment in an enterprise production environment, and availability of third-party applications and components.

The source code for both the Django and the Rails applications are available in a subversion repository at http://3columns.net/habitual/.

1.1 Specification Overview

The small application the authors implemented is named “Habitual Readers.” It purports to be the public website of a book reading club, allowing viewers to see which club memebers have read which books, and their comments upon each read book. The application also categorises books using tags, and retrieves information from Amazon [*http://www.amazon.com/ -Ed*]. Along with the specification, the authors also developed a static HTML prototype of each of the application's seven public pages. The basic HTML and CSS layout is a modified version of a template by Andreas Viklund [VIKLUND].

The Habitual Readers specification requires that the application allow logged readers to add, change and delete books, readers, reading occasions (i.e. An occasion when a reader read a book) and tags. It does not specify the look and feel of these “admin” pages.

1.2 Habitual Readers Implementation

We developed the software on our own PCs, on a part-time basis, recording the time spent on each development task. We each used the latest version of each web framework - “Edge Rails” for Ruby and “SVN trunk” for Django – as of August, 2006.

Approaching the implementation of this application, Ben and Alan had approximately equivalent experience in Rails and Django. Ben has been working in Ruby and Ruby on Rails on a part time basis for six months. Ben has also been developing in various database-backed software environments since the earliy nineties. Alan had only two months of part-time Django experience, though he had been developing web applications in various Python frameworks since 2003, and in other languages since 1997.

At the conclusion of the implementation, we noted the following variation from the specification:

1. The Django application includes an extra “/” at the end of each URL. This behaviour cannot be easily changed without affecting the Django admin application.

2 Quantitative Comparison

We compared the two application implementations using two convenient measures: lines of code and implementation time. These quantitative measures were then used to inform the analysis that accompanies the qualitative comparisons below.

2.1 Lines of Code

In producing these counts, we included all files containing a large proportion of hand-written code. Rails and Django generate configuration files in their native language – Ruby or Python – which the developer is required to modify. We did not include these configuration files. Python and Ruby lines of code were measured using David A Wheeler's “sloccount” tool [WHEELER], which ignores blank lines and comments. HTML line counts were produced by running the “wc” shell command on each implementation's template files.

	Rails	***Django***
Ruby / Python Model Code	83	116
View / Controller Code	203	109
HTML Helpers/ Template Tags	56	26
Schema Migration	118	-
YAML Data Loading	26	69
Authentication	31	-
Ruby / Python subtotal	517	320
Templates	297	406
Totals	814	726

Comparing primary implementation languages, the Django application contains one third fewer lines of code than the Rails application. The difference would have been larger if we had not implemented YAML data loading in the Django implementation, as the YAML data loading counts for 20% of the lines of code in the Django application. (YAML is “Yet Another Markup Language”, which has a syntax convenient for specifying test data [YAML]).

There is also a large difference in the number of lines of template code. The Django templates have one third again as many lines as the Rails templates, even though the Rails templates encompass the admin pages while the Django tamplates do not.

2.2 Implementation Time

The authors recorded the time they took to implement the Habitual Readers application, with the results presented in the table below. We have recorded time in three columns: time spent on the Rails implementation, time spent on the Django implementation, and time spend on tasks that benefit both projects. All times are measured in elapsed hours and minutes.

Task	*Rails*	*Django*	*Common to Both*
Initial HTML Protoype			4:30
Develop Test Data			1:36
Project Set Up	0:15	0:11	
Models	1:30	0:09	
Home Page	3:00	1:40	
Basic Pages	5:00	2:08	
Admin Pages	8:25	:57	
Amazon Interface	1:00	2:18	
Data Loading (Code)		1:36	
Test, Review and Tidy	1:30	1:31	
Totals	**20:40**	**10:30**	**6:06**
Totals including “Common times”	**26:46**	**16:36**	

It was clearly faster for Alan to implement this application in Django than for Ben to implement this application in Rails. The bulk of this difference can be attributed to the extra effort required to implement the administration functions in Rails which accounts for approximately seven and a half hours. Excluding the admin pages, and allowing for factors such as the variability in work environments and the experience of each developer, the implementation times are approximately equal.

3 Qualitative Comparison

This section compares a selection of attributes of Rails and Django.

3.1 HTML Templating

HTML templates are a core feature of both Rails and Django. Both allow templates to be composed of a base template that defines the overall layout of the page, plus individual page templates that define the specific content of the page. In Rails, the base template is called a “layout” while the individual page templates are “views”. In Django, both are plain templates.

The key difference between the framework is the way they embed dynamic content. Rails views use in-line Ruby code fragments, and may therefore contain arbitrarily complex functionality. Rails encourages developers to be as pithy in their view code as they are in their controller or model code, rewarding aggressive refactoring with a satisfyingly concise result. Django, in contrast, uses a simpler templating language conceived specifically so that web designers with HTML skills, but minimal programming knowledge, can build templates.

For example, on the main page of Habitual Readers, there is a tag cloud – a list of links to tags, sized depending on the popularity of the tag. The Rails application retrieves the list of tags from the database and makes it available to the view. The section of the view that renders these tags is:

```
<%= render :partial -> 'tag cloud' %>
```

This refers to a Ruby partial, a fragment of a page, which is contained in `_tag_cloud.html`. That file's content is:

```
<h3>categories</h3>
<p class='taglist'>
      <%= render :partial -> 'tag', :collection => @tags %>
</p>
```

This generates the heading, then refers to another partial, which will be used once for each object in the collection named `tags`. The `_tag.rhtml` file contains a single line:

```
<%= link tag[:name], tag_url(tag[:name]), :class => “level_#{tag[:pop_level]}”
%>
```

This call a Ruby helper function to generate a link to the tag, setting the CSS class appropriately.

By contrast, the portion of the Django template that generates the output is:

```
<h3>categories</h3>
<p class=”taglist”>
{% for tag in tags %}
      <a class=”level+{{ tag.pop_level }}”
            href=”{{tag.get_absolute_url }}”>{{ tag.name }}</a>
{% endfor %}
</p>
```

The Rails views appear more complex than the Django templates, spreading the HTML template to generate the tag cloud across three separate files. The advantage of the Rails approach is that it allows the developer to reuse even the smallest fragment of code, resulting in significantly less repetition and fewer overall lines of template code, even on a small application such as Habitual Readers. Rails doesn't force this approach, however, allowing the developer to produce a near identical view to the Django template as demonstrated in the following example:

```
<h3>categories</h3>
<p class=”taglist”>
<% @tags.each do |tag| %>
      <%= link_to tag[:name], tag_url(tag[:name]), :class => “level_%#{tag:\
            popup:level]}”
%>
```

```
<% end %>
</p>
```

The more explicit approach taken by Django is considered simpler to teach to web designers without any background in software development [CROFT].

3.2 Schema Generation and Evolution

Model objects are a central concept in both Rails and Django, through each take a different approach in their implementation. Rails also provides facilities for schema evolution, which Django does not.

3.2.1 Defining Model Classes

Rails implements the active record patterns [FOWLER, M] as its data access layer. To define a model, the developer derives a new class from the ActiveRecord base class. ActiveRecord deduces the model's attributes from the table in the database that matches the plural of the class name. The developer can modify this mapping, but it is generally considered good Rails practise to keep such modifications to a minimum.

Django, on the other hand, requires the developer to explicitly specify not only each class, but also each attribute. Django has the standard tools for creating a database schema from an application's model definition.

A model class from the Habitual Readers application serves as an example. Instances of the ReadingOccasion class record an occasion when a reader read a book. Django code explicitly defines each attribute of the model, including metadata necessary to define the underlying database table.

```
class ReadingOccasion(models.Model):
      reader = models.ForeignKey(Reader)
      book = models.ForeignKey(Book, edit_inline=models.STACKED,
            num_in_admin=1)
      finished = models.DateField(core=True)
      reading_time = models.FloatField(core=True, max_digits=5,
            decimal_places=2, blank=True)
      notes = models.TextField(maxlength = 2000, blank = True)
```

The Django model also includes a small amount of metadata used to specify its appearance in the automatically generated admin pages, which are explained below.

By contrast, the Rails version is minimal. It defines only the relationships between itself and other models. All of the attributes of the model (when the book was read, how long it took to read and additional notes) are added when Rails examines the table definition at runtime:

```
class Reading < ActiveRecord::Base
      belongs_to :book
      belongs_to :reader
end
```

In comparing the two, we see that the Rails class is much shorter, with the trade off being that the model's attributes are not documented in the class definition. Django, on the other hand, requires that the developer define the entire model, including its attributes, within the class

definition. This is another example of Rails choosing the concise over the explicit while Django has chosen the converse.

3.2.2 Evolving Model Classes

While Django has facilities for creating a database schema from the model definition, it does not provide the developer with any support for modifying (or, “evolving”) the model definitions while preserving data in the underlying database tables. The Rails migrations mechanism addresses creating and evolving model classes and the underlying scheme while preserving data.

A migration is a Ruby script that defines additions, modifications and deletions to the database schema. Optionally, the developer can also specify how data is to be migrated from the old version of the schema to the new version. Each migration script is assigned a version number, and changes to the schema can be rolled backwards and forwards by applying and reversing the migration versions in sequence.

Here is the migration used in the Rails Habitual Readers application to create Readings table:

```
class CreateReadings < ActiveRecord::Migration
      def self.up
            create_table :readings do |t|
            t.column “book_id”, :integer
            t.column “reader_id”, :integer
            t.column “date_read”, :datetime
            t.column “reading_time”, :integer
            t.column “notes”, :text
      end

      def self.down
            drop_table :readings
      end
end
```

The `up` method is invoked to apply the migration. It will create a table named `readings`, with the given columns. The `down` method is invoked to reverse the migration – in this case dropping the `readings` table. In understanding migrations, it is important to note that the only effect of the migration script is the modification of the schema when the script is run. When the Rails application is executing, the model class reads the database schema to dynamically determine its attributes at runtime.

There are two key advantages to Rails' incremental migration compared with Django. First, Rails provides a standard mechanism mfor deploying new releases to already running production systems while preserving data. For example, if a databas column's type is changed from char to integer, the accompanying Rails migration script would specify the steps required to move the data from the old char column to the new integer column. To perform similar operations in Django, the developer would need to write an ad-hoc SQL script.

The second advantage is that being easily rolled back, migration encourages a certain amount of experimentation with the model classes and database schema. Certainly some experimentation with models is possible in Django, especially if the model code is kept under source code control. However, as data is not preserved through such changes, it is less attractive unless there is a mechanism for quickly loading test data.

At the time of writing, the Django development community is working toward introducing a schema evolution mechanism.

3.3 Automatically Generated Admin Pages

Many web applications have a group of “admin” pages, pages used by a small number of trusted users, perhaps to enter content for publishing to a wider audience, or maintaining reference tables.

A clear area of difference between Rails and Django is Django's automatincally generated admin pages, which can save a significant proportion of development time. For example, in the Habitual Readers implementations, the development of admin pages in Rails took 29% of the development time, compared to 6% for Django.

When we write that the Django admin pages are “automatically generated”, we mean that Django generates them with only small hints from the developer. The developer gives these hints by adding attributes and parameters to the Django model classes and their fields. Despite being a little fiddly, this process is rather quick and the generated pages are suitable in a wide range situations. To further customise the look and feel, the developer may provide Django with alternate templates.

Rails chooses not to implement an administrative interface based on a distinction drawn between application infrastructure and application functionality. There are a number of Rails plugins that aim to fill the same need as the Django admin application. Two such plugins are AutoAdmin [AUTOADMIN] and Streamlined [STREAMLINED] (both in the early stages of development). Developers using Rails in their own projects are advised to check on the status of these plug-ins before commencing projects that require an administrative interface.

3.4 Internationalisation

Although we did not address internationalisation with the Habitual Readers application, this is a well-documented and much-discussed issue in both the Rails and Django communities.

In its current release, some Ruby core libraries, for example String, aren't unicode aware. This means that string operations such as length and split won't function as expected when working with non-Western text. The Rails core team have addressed this problem by merging the multibyte_for_rails plugin with the Rails ActiveSupport module as ActiveSupport::Multibyte [RAILSMULTIBYTE]. The upshot of this is that as of version 1.2, the Rails framework will extend the problematic Ruby core libraries to make them unicode-aware.

Python has had Unicode support and GNU gettext [GETTEXT] style internationalisation since version 1.6, which was released in 2000. The Django framework builds on this with standardised mechanisms for internationalising Django applications and localising responses for individual HTTP requests.

3.5 Integrating Third Party Code

Rails and Django can each take advantage of two flavours of extensions. The first is what Rails calls “plugins” and Django calls “applications”. This kind of extension is aware that it is running inside of its respective framework. The second is third party libraries written in Ruby or Python, but that aren't specifically developed for use within either framework.

Rails plugins are discrete collections of code that can be included within a Rails application. The Habitual Readers Rails implementation took advantage of the acts_as_taggable plugin to provide support for tagging books, and the acts_as_attachment plugin to support uploading a reader image and resizing it to a manageable thumbnail for use in the application. At the time of writing, 419 Rails plugins are available for download from http://www.agilewebdevelopment.com/.

Django applications can be used in a manner similar to Rails plugins. A Django site may be composed of several applications, each of which may contain model code, templates, template tags and plain Python libraries. Typically one application will contain the site's main functionality with additional applications providing features such as comments and admin pages. At the time of writing, there is no central repository of re-usable, third-party

applications.

Rails and Django freely integrate with a wide range of native libraries – that is, libraries written in Ruby and Python respectively. For example, in the Habitual Readers application, both the Rails and Django implementations communicate with Amazon via native third-party libraries.

3.6 AJAX

While neither the Rails nor the Django implementations of the Habitual Readers application took advantage of AJAX, it has become so prevalent in web applications that it's worth mentioning here.

Rails includes a number of helper functions that assist with sending and responding to XmlHttpRequests. Using RJX, a Rails developer can write a template that responds to a request by generating Javascript that will be sent back in the response object and executed in the browser. Prototype and Scriptaculous helper methods are available that allow functions from those Javascript libraries to be used in RJS templates. The packaging of these Javascript libraries with Rails does not preclude the developer from choosing to work with Javascript libraries.

In contrast, Django includes only a JSON module, leaving Javascript code and the choice of a Javascript library, if any, to the developer. The Django core team argue that this is a strength of Django. Other Django developers, however, have indicated that they would prefer the Rails approach of an officially sanctioned Javascript library and associated helper functions.

3.7 Other Considerations

In addition, the following non-technical areas may be of concern to developers choosing between the two frameworks.

3.7.1 Maturity

Both frameworks were extracted from web applications that were developed in the 2003-2004 period. Rails was released to the public in July 2004, and Django in July 2005. As such, Rails has had a head start in getting community contributions to the framework and reached the milestone 1.0 release in December 2005. The current release of Django is 0.95, and there may be further changes to the Django API before the 1.0 milestone release is reached [DJANGOAPI]. There are currently 12 core team members who have commit rights on the Rails repository [RAILSCORE], and 4 on the Django repository [DJANGOCORE].

3.7.2 Market Position and Hype

A search on job sites JobServe [JOBSERVE] and Seek [SEEK] shows Rails turning up in job requirements more often than Django at a ratio of 6:1, while Python shows up more often than Ruby at a ratio of approximately 4:1.

While the Python language has a higher demand than Ruby, the Rails framework is more firmly established in the marketplace than Django. Compared to Java and J2EE, Rails and Django are both young when it comes to gaining acceptance in the marketplace.

Similar comparisons can be made using tools such as Google Battle [GOOGLEBATTLE], Ice Rocket [ICEROCKET] and Technorati [TECHNORATI] where Rails consistently comes out on top in the hype stakes.

3.7.3 Tools and Utilities

Both frameworks ship with scripts that assist in the development process and automate repetitive tasks such as schema creation and executing unit tests. Rails takes advantage of Rake, the Ruby build language to automate repetitive tasks.

Capistrano is a deployment tool that has been developed for automating the deployment of Rails applications. It executes commands on remote servers via ssh to perform tasks such as checking the latest revision of code out of a repository, running migrations and loading data. It is particularly useful when deploying to multiple production servers as it can execute commands on multiple servers in parallel. At this stage there is no such tool for Django, although Capistrano has been used by others to deploy non-Rails applications, so there seems no reason that it couldn't be used to deploy a Django application.

4 Conclusion

Django and Rails aim to solve similar problems, in a similar manner, using a similar architecture. There is no clear technical benefit for an experienced Rails development team to switch to Django or for an experienced Django development team to switch to Rails. For developers not currently working with either Django or Rails, the most important consideration is the implementation language. Ruby developers would benefit from using Rails, while Python developers would benefit from using Django, allowing them to apply skills they already have.

For developers who know neither (or both) languages, the “best” framework will depend on the development environment and type of application. The following table summarises those aspects that we have investigated in this paper:

Factor	***Rails***	***Django***
Support for model and schema evolution	Integrated framework for schema evolution.	Minimal.
Internationalisation	Some support in Rails 1.2.	Some support.
Designer friendly templates?	Possible, with the use of a third-party library.	Yes.
Third party plugin support?	Mature plugin architecture, well-used by the community.	Some support via the applications mechanism.
Javascript Support	Prototype and Scriptaculous bundled with Railes. RJS framework simplifies their use.	Possible, but no direct support for any particular library.
Flavour	Concise.	Explicit.

While choosing between these two frameworks may be difficult, the good news is that either framework is a good choice for a team wishing to develop a web application.

5 References

1. [AUTOADMIN] – http://code.trebex.net/auto-admin
2. [CROFT] – http://www2.jeffcroft.com/2006/may/02/django-non-programmers/
3. [DJANGOAPI] – http://www.djangoproject.com/documentation/api_stability/
4. [DJANGOCORE] – http://www.djangoproject.com/documentation/faq#who-s-behind-this
5. [FOWLER] – Rails Recipes, Chad Fowler
6. [FOWLER, M] – Patterns of Enterprise Application Architecture, Martin Fowler
7. [GETTEXT] – http://www.gnu.org/software/gettext/
8. [GOOGLEBATTLE] – http://www.googlebattle.com/
9. [ICEROCKET] – http://www.icerocket.com/

10. [JOBSERVE] – http://www.jobserve.com/
11. [PYAMAZON] – http://www.josephson.org/projects/pyamazon/
12. [RAILSCORE] – http://rubyonrails.org/core
13. [RAILSMULTIBYTE] – A Rails Unicode primer, http://fngtps.com/projects/multibyte_for_rails/wiki/UnicodePrimer
14. [RSPEC] – http://rspec.rubyforge.org/tools/rails.html
15. [RUBYAMAZON] – http://www.caliban.org/ruby/ruby-amazon.shtml
16. [SEEK] – http://www.seek.com.au
17. [STREAMLINED] – http://streamlines.relevancellc.com/
18. [TAGGABLE] – http://wiki.rubyonrails.org/rails/pages/ActsAsTaggablePluginHowto
19. [TECHNORATI] – http://www.technorati.com/
20. [VIKLUND] – http://andreasviklund.com/templates/
21. [WHEELER] – David A. Wheeler's Sloccount tool, http://www.dwheeler.com/sloccount
22. [YAML] – Yet Another Markup Language, http://www.yaml.org/