Evaluating Web Development Frameworks: Rails and Django

white paper by

Jonathan S. Linowes
Parkerhill Technology Group LLC
February 26, 2007 (Last update: March 22, 2007)

This white paper is a technical manager's guide to evaluating web development frameworks, with a detailed review of Ruby on Rails and the Django (Python) projects.

In this study we establish a list of evaluation criteria and then discuss and rate how well each framework addresses each criteria.

As your priorities and assessment may differ from ours, this report can be used as a road map for your own evaluation.

If you have already decided to use a specific framework or recently joined a team, this report may serve as a high level introduction, providing valuable context for your learning curve.

Table of Contents

1. Introduction

Anyone developing a dynamic web site these days should at least consider (if not require) a web development framework which provides a layer of abstraction and provides shortcuts for frequent programming tasks. A generally accepted architecture paradigm for web frameworks is the model-view-controller (MVC) which separates the content data structure (model) from the presentation of this content (view), and the handling of user requests for this content (controller).

There are many good MVC frameworks available for web development, written in various programming languages, with subtle (and not so subtle) technical and conceptual differences. The one creating the most buzz and attention is Rails (written in the Ruby language). Another project gaining momentum is Django (written in Python). Both of these are true MVC architectures.

Another approach to web development is to use a higher level, more complete web application, such as content management systems (CMS), which have a great deal of functionality already built in. The Xaraya project, for example, adheres to the MVC paradigm, has hook-able CMS level modules, and allows easy custom layouts and templates. And with its advanced data modeling (DynamicData) Xaraya also reaches into the framework arena. Having used Xaraya successfully for more than two years, I recommend it as part of anyone's website development arsenal.

However, we at Parkerhill Technology Group increasingly find the need for a lower level, lighter, framework for a number of projects. We have narrowed our alternatives to Rails and Django. This report presents my analysis. I have tried to accurately and objectively describe details of each project (especially those that matter most to us), and then give my analysis and opinion separately.

Evaluation Criteria

There are many factors to consider when evaluating a web development framework for your application development project.

Our perspective is that of a web developer, project manager, and entrepreneur, with specific needs that may vary from one project to the next. Our criteria and priorities certainly may not align with yours. Furthermore keep in mind this is a moving target, both Rails and Django frameworks are young, improving continuously, and gaining non-core contributions from their respective communities.

Disclaimer: While I come at this with a significant level of experience, this report is the result of my review without the benefit of having built a real project with either framework.

The following chart outlines my criteria which will be rated on a scale of 1 to 5 (1=worst, 5=best). The final results will be presented in the last section.

Evaluation Criteria	Rails	Django
TECHNICAL		
Programming Language		

Evaluation Criteria	**Rails**	**Django**
Framework Concept		
Directory Structure		
Database / Model		
URL Routing		
Controllers / Views		
Templates		
Forms		
Data Administration		
User Administration (authentication, sessions)		
AJAX		
RESTful		
RSS/Atom/XML		
Internationalization / Localization		
Caching		
Security		
TECHNICAL SCORE		
SUPPORT		
Project / Community		
Documentation		
User Extensions		
Development / Debugging		
Test Tools		
Deployment Support		
SUPPORT SCORE		
OVERALL SCORE		

Let the Flames Begin!

So let's get the emotions out of the way first. There's no avoiding it, some people are very passionate about their favorite technologies. I've read many blog postings comparing Rails and Django (plus the dozens of comments they generate). Some people are firmly positioned in one camp or the other. Some people say "Hey, they're more similar than different so just choose what you prefer". And others have done some sort of honest comparative analysis. If you're interested, do a Google search on “rails django”. Here are a few good ones:

- Video of debate between Adrian Holovaty, creator of Django, and David Heinemeier Hanson, creator of Rails: http://www.djangoproject.com/snakesandrubies/ (Dec 3, 2005)
- A simple quantitative comparison by implementing a small project in both frameworks http://docs.google.com/View?docid=dcn8282p_1hg4sr9
- A detailed analysis, a bit out of date: http://magpiebrain.com/blog/2005/08/14/a-comparison-of-django-with-rails/ (August, 2005)
- And here's one comparing performance: http://wiki.rubyonrails.com/rails/pages/Framework+Performance
- Someone's personal experience, for what it's worth: http://blog.carlmercier.com/2007/01/30/why-i-moved-from-ruby-on-rails-to-pythondjango-and-back/
- This is funny, kind of sums it up for some people: http://www.djangoproject.com/weblog/2006/dec/06/comparisons/

2. Project and Community

Status and History

Both Rails and Django projects are relatively new, open-source projects. Both emerged from real world websites, proven to handle very large volumes of hits, and encourage agile development (quick, incremental updates) of your web projects.

Rails was developed by 37Signals Inc. (Chicago, Illinois) for a collaborative project management application called Basecamp. Rails was released as open source in July, 2004, released version 1.0 in December, 2005. The current version is 1.2.1, recently released in January, 2007.

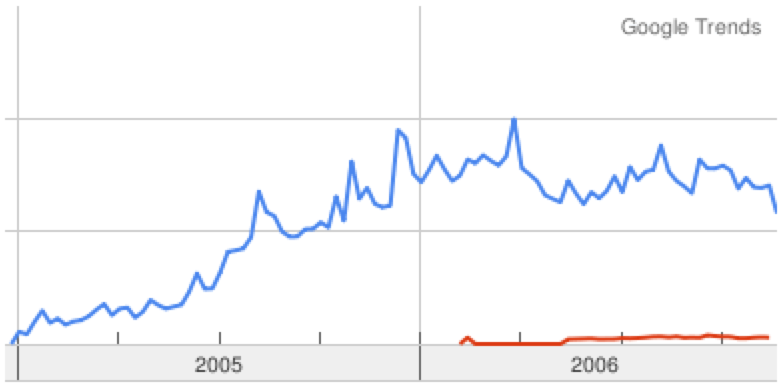
Django was developed by Lawrence Journal-World , a small town newspaper in Lawrence, Kansas, for their innovative family of websites. Django actually began development a bit earlier than Rails (2003), but wasn't released as open source until July 2005, and hasn't reached Release 1.0 yet. Normally a 1.0 means the API's (core methods) are stable, and features are finalized and frozen. (To track this, see http://code.djangoproject.com/wiki/VersionOneFeatures, and http://www.djangoproject.com/documentation/api_stability/).

In mid-2006, Django went through a major refactoring-- that is, an overhaul of its internal coding to clean it up old stuff, make things more consistent, efficient, et cetera. Rails, I have heard, has not yet been refactored, except for one or two components.

Both projects sport a respectable list of commercial, high volume sites using their technologies.

Project Community

The Rails community is decidedly the larger of the two. There are lots of ways to show statistics on this. Google Trends graphs compare the frequency of Google search terms: "ruby rails, django python"



Chat room (irc) activity is another way to see project activity (spot check on 2/15/07): #rubyonrails: 486 members; #django: 219 members. Similarly, Google groups membership: rubyonrails-talk : 7485 members; django-users : 4180 members.

Clearly Rails is the big guy and enjoys "first-mover advantage". Some say they're just really good at marketing. Good early documentation and online screencast demos helped tremendously. There are several books on developing with Rails (including version 2 of the "Agile Web Development with Rails" book). The annual RailConf conferences are fairly well attended (hundreds, not thousands, of attendees).

Django too has a strong active community. Emerging from the Python world, it's already started to create some good buzz. They too have some screencasts, and The Django Book is in beta draft form. Django folks gather at the annual PyCon conference, where in 2007 Django is one of the tracks.

Both Rails and Django use Trac for feature and bug reports, a wiki-based issue tracking system for software development projects. Both use subversion for source code repository. The community participates by making feature requests, reporting bugs, writing documentation and tutorials, and providing useful re-usable code. That's the beauty of open-source software projects.

Opinion

Considerable value must be given to the size and momentum of Rails and its community. They have very smart and passionate people contributing code, plug-ins, tutorials, and support. (Django does too, only Rails has more of them!) .

But a word of caution. With popularity also comes more "newbies" - new users, many being inexperienced not just with the language and framework, but perhaps any kind of modern programming. In some ways this is great. But it can potentially be a drag on the quality and responsiveness of the project. If/when the Django community grows, it will face the same problems but right now its following seems to be mostly experienced programmers.

I have hung out and asked questions in both projects' irc channels (chat rooms). The people in both are friendly, helpful, and available.

Django is clearly behind, having had a later start, is not yet at Release 1.0, its book in Beta. If you're not comfortable building on a pre-1.0 product, then your decision is easy. Otherwise, don't worry, as it is not unusual for the idea of "no wine before its time" be used in OSS (open source software) projects. The fact remains there are plenty of production sites using Django in its current condition.

Conclusion: Both projects are active and vital. Rails is bigger and stronger.

Project / Community	Rails	Django
When open sourced	July 2004	July 2005
When 1.0 released	Dec. 2005	TBD

Project / Community	**Rails**	**Django**
Current release (as of Feb 15, 2007)	1.2.1	0.95.1
Repos	subversion	subversion
Issue tracking	trac	trac
Website, wiki, blogs	very good	very good
irc channel members (Feb 15, 2007 3pm EST)	486	219
Google Groups membership (Feb 15, 2007)	7485 (rubyonrails-talk)	4180 (django-users)
MY RATING: (1=worst, 5=best)	**4**	**3**

3. Programming Language

It could be argued that the most telling difference between Rails and Django is the language they're implemented in., and the culture that has built up around the languages' communities.

Language Community

Here are some rough metrics on the popularity and communities of the the Ruby and Python languages (and Php for comparison).

	Ruby	**Python**	**Php**
Year invented	1993	1991	1994
TIOBE Index (Feb, 2007)	10th (2.5%) (up from 20th Feb 2006)	7th (3.6%)	4th (8.8%)
Users Groups, USA	94	32	
Chat room members irc.freenode.net (Feb 15, 2007)	373 (#ruby-lang)	357 (#python)	330 (#php)
Freshmeat projects (www.freshmeat.net)	327	2442	3798
Frameworks (sampling)	Rails, Nitro, Merb	Django, TurboGears, Pylon	CakePhp, Symfony

The TIOBE Programming Community Index (http://www.tiobe.com/tpci.htm) is a rough indicator of the popularity of major programming languages (the top 3 languages being Java, C and C++ respectively, FYI).

Both Php and Python have more history and a much wider breadth of usage than Ruby. Python, for example, is known to be used in mission critical projects at NASA, an indication of its quality and reliability.

The Ruby language, a relative newcomer in the West, was invented in Japan. In the US, most people know of (and use) Ruby in context of the Rails framework. Although many Ruby champions might strongly disagree with this statement.

Language Features

It is not my intent to directly compare the language structure, syntax and semantics. I will only share some impressions.

Ruby and Python are really quite similar. For example,

- Both are strong object-oriented languages. That is, everything is an object.
- Both are dynamic languages, lets you write code that writes code.

- Both are dynamically typed (because it doesn't use explicit data-type declarations) and strongly typed (because once it has a data-type it actually matters)
- Both have a compact syntax.
- Both are quite easy to read, which is important when going through other people's code (or even your own a few months later).
- Neither requires braces around bodies of code (Ruby uses “end” to indicate the end; although you can use braces if you want. Python relies on leading whitespace indentation).
- Both have an interactive command-line console.
- Both have testing infrastructures
- Both have doc generators (from comments in the source code)
- Both pride themselves on being a generation ahead of languages like C, Php, and Java

Ruby

Ruby has an especially compact syntax. No wasted words, that's for sure. It is said that Ruby's program statements read more like a natural language (or rather, a domain-specific language) than a general programming language. People pride themselves on how small they can reduce a block of code, like solving a puzzle. (And part of the challenge is to make the resulting code even more readable than the more verbose version; you certainly don't want to make it cryptic). This helps make programming in Ruby “fun”, a word people seem to associate with Ruby and Rails development. Common programming patterns (idioms) have become part of the Ruby culture.

Some cool things about Ruby: Don't need parenthesis on function calls. Dynamic blocks are like functions 'on the fly' , which can be used for example, with iterators and lists. (For example, `('a'..'e').each {|char| print char }` evaluates to “abcde”).

RubyGems is the tool for downloading Ruby packages.

Python

> “Python code seems to be distinctive in having a particular character, and the Python community seems to have a lot less Cowboy Programmers than most other languages. If you hate to conform to standards, you won't like Python I guess.” (http://mail.python.org/pipermail/europython/2005-April/004995.html)

Python has a strong following as a general purpose programming language. It is object-oriented, but you can use it procedurally if you want (Django doesn't). Python has a clearer (or perhaps just more formal) syntax than Ruby. Indentation (leading whitespace) is significant, to group statements.

Python programmers use the endearing term “*Pythonic*” to describe code that adheres to the Python philosophy. See http://en.wikipedia.org/wiki/Python_philosophy (it's also hidden within Python itself, enter: `>>> import this` at the Python prompt).

Python has a large selection of third-party libraries available and is used in a wide range of domains, including the engineering and scientific communities. Python has the setuptools package manager (amon others). There are over 10,000 libraries listed on the setuptools site.

Opinion

Ruby is a bit looser in how you express yourself, like its idiomatic expressions and dynamic blocks. In that, Ruby does seem more “fun”. Some people take to Ruby immediately.

Other people are really bothered by this aspect of Ruby, believing a programming language should be more concise, preferring that when there is one best way to express something, your code will be more clear, less prone to bugs, and more portable between programmers.

Ruby programmers who have used Python say they were grateful to leave behind little “annoyances”, like parenthesis in function calls, having to declare the “self” argument to methods, and the special use of “__” (double-underscores) (e.g. __init__ for constructors). Python users defend these as non-issues. I suppose when little things like this bother you, it can grow increasingly annoying, until you finally want to kill someone, like in an Edger Allen Poe story (well, maybe that's an exaggeration).

You certainly don't want your language to be tedious. It doesn't look like either Ruby or Python are that. Far from it. But otherwise how important is this “fun” thing anyway? Seriously. (no pun intended). Ultimately the fun is in building websites that work, reliably. And I have no doubt, whichever language you use, the pain and frustration will still come when things don't work the way you expect (as I've witnessed on both framework's chat channels).

Conclusion: Both languages seem equally complete and flexible. Python is more mature and explicit. Ruby has that fun-factor. I rate them both a "4" (out of 5), for different reasons.

4. Framework Concept

Conceptually both Rails and Django are very similar, adhering to the model-view-controller architecture. I will not go into details here about MVC, as there are plenty of references not the least of which are the introductory chapters to the Rails and Django documentation and books, and Wikipedia.

However it's important to say that Django is not a Rails clone. And there are quite a few Rails-clones around, in any number of languages. Rather, Django and Rails have important differences both in concept and implementation.

Conceptual Commonality

First, the common ground, very briefly. Both Rails and Django adhere to modern principles of

- Object architecture – use object classes and inheritance to organize and encapsulate data and logic;
- Agile development-- the ability to implement small, incremental changes and features quickly; agile development encourages collaboration;
- DRY – stands for "don't repeat yourself," create reusable snippets of code rather than duplication via copy/paste.

Both projects agree that a framework should provide:

- Allow easy modeling of your application's data, and encapsulate the business logic into objects, including responsibility for create, retrieve, update and delete (CRUD) of these objects in a database;
- Map user events (like URL requests from a browser) to application code that accesses the model and generates a response (view) back to the user's browser;
- Use a separate templating system to define how data is presented to the user, whether html+css layouts and styles, or some other data formats (XML).

In addition, both projects agree that the framework should:

- Support test-driven development
- Support generated documentation
- Support community driven enhancements

Both frameworks provide

- Scripts for creating a new project with default files
- An embedded web server for development

- more...

In general, both frameworks provide a program "stack" -- layers of functionality, written for the framework and tightly integrated (as opposed fully independent components glued together).

Conceptual Differences

Rails

Rails promotes the idea of "opinionated software" and "convention over configuration", where the system automatically builds and fits the initial pieces together for you. Then you can accept these reasonable defaults or modify/add just what is necessary, incrementally, as you build your application.

Rails also advocates "constraints are liberating". For example, the directory layout is well defined and your application must conform to this convention. The idea is that when decisions like these are made for you, then you as a developer do not have to spend precious brain cycles making these basic decisions, and can focus instead on what's really important-- designing and implementing your website. It also facilitates team collaboration because new players will more readily know, for example, where to find files and know what they do based on their names.

Another example is naming conventions like *automatic pluralization*. Let's say your site manages products-- in Rails, your database table is named with the plural "Products", and your object class is singular, "Product". Rails handles the pluralization/ singularization automatically. This is handy and makes things more intuitive as you construct your application. Of course, you can override these defaults if needed.

One more example is AJAX support. Rails has standardized on a common AJAX library, provides conventions for server side scripting (.rjs files), and allows you to integrate AJAX into your apps while programming in the Ruby language and not even seeing the Javascript.

In this way, Rails takes pride that many things are done "auto-magically". In Rails, like in life, fun is magical, and magic is fun!

Django

One of the driving principles behind Django is "loose coupling", where each aspect of the framework tries to remain independent of the others. While all the pieces work together in tandem, they are not dependent upon each other. This makes it fairly easy, in most cases, to substitute alternative components if desired or necessary. More important, it forces a discipline of clean orthogonal architecture.

For example, URL routing is completely decoupled from the view methods that get called.

Another example is how Django does not require you adhere to a strict directory structure. They suggest an organization (as built with the but you can put and move things wherever you need or desire.

Django prefers transparency over "magic". In 2006 the project underwent a refactoring (rewrite) where implicit features were removed. This was blatantly referred to as "magic-removal" (for a complete list of changes in that effort see http://code.djangoproject.com/wiki/RemovingTheMagic). For example, Django, like Rails, originally had automatic pluralization, but this was taken out.

With regard to AJAX, Django intentionally does not encapsulate AJAX into its framework. Rather, it provides *support* for AJAX in how data is passed between server and client (serialization), but lets you choose any of the dozens of good AJAX toolkits available. More loose coupling and being explicit.

Despite what I said earlier about both Rails and Django being MVC (model-view-controller) architectures, Django prefers the MTV acronym -- *model-template-view*. That is, Rails *controller* classes are equivalent to *views* in Django, and Rails *views* are what Django simply calls *templates*.

Opinion

> "Any sufficiently advanced technology looks like magic" – *A. C. Clarke*

Or,

> ma·gi·cian -- *[muh-jish-uhn]* -- an entertainer who is skilled in producing illusion by sleight of hand, deceptive devices, etc.; conjurer. (Dictionary.com).

I really appreciate Rails' efforts to take care of the busy work for me, and to encapsulate complexity (like Javascript) that I'd really rather not have to do myself if I don't have to (provided I can get under the hood when necessary). In fact, that's a great way to get going quickly and learn how things work at the same time.

On the other hand, I really appreciate Django's efforts to keep everything clean, decoupled, and transparent. That helps keep the guess work out of programming. It probably also helps with debugging, where questions like "is the problem something I wrote, or just how I'm using the framework?" might be easier to discern in Django.

So perhaps the tide shifts depending whether you put more weight on efficient development versus efficient debugging / maintenance. (I may be extremely wrong about this, it's just a hunch).

I've heard Rails-skeptics caution that the framework could (will) become bloated in a few years (bloated means big, slow, clunky), likened to the enthusiasm Php experienced in the 1990's.

Django may require me to learn web technology more deeply to use the framework effectively. This is probably a good thing, we all should know our craft. Then again, we can't all be expected to know everything, and still get work done. Hmm...

OK, I'm going to get killed for this one-- *Rails reminds me of Windows. Django reminds me of Linux.* Sure both are open-source, I'm not talking about that. But Windows took an early emphasis as "the any-man's operating system", magically hiding a lot of technical details, that's great for many people, and very frustrating for others. Likewise, Linux is composed of many loosely coupled components, is very transparent, is lean, reliable, and frustrates the heck out of normal people. Maybe one day we'll get the

"OSX of Frameworks" which marries the best of both? :)

Conclusion: I rate Django higher, with a score of 4, versus Rails at 3. I prefer the loose coupling and explicit design over a more paternalistic, opinionated, magical approach.

Framework Concept	Rails	Django
Policy	paternalistic, opinionated	loose coupling
Transparency	"magic"	explicit
MY RATING: (1=worst, 5=best)	**3**	**4**

A few words about this rating: Unlike most of the other criteria, my rating here is much more a gut feel than a technical or quantitative assessment. I simply wanted to give Django a higher score for the reasons cited above. From the beginning, I decided not to do fractional values. And I couldn't justify giving either a score of 5, since there's always room for improvement and new concepts will emerge that build on current experience.

5. Installation / Directory Structure

I am not attempting to provide a manual for the frameworks but I do believe that looking into the generated file system directory structure can provide some useful insight.

With regard to installation, my experience with each was comparable and pretty uneventful. I did blog the details at:

- Installing Rails on my iMac
- Installing Django on my iMac

Note: Ruby files have the .rb extension, Python files have .py.

Both Rails and Django come with a lightweight, single process web server for development in your local environment.

Rails

To start a Rails project you use the command:

```
$ rails myproject
```

After generating a new project you'll find a full and rich set of subdirectories all ready for you to play in.

```
myproject/
      app/                    your application files
            controllers/            your controller classes
            helpers/                your helper functions
            models/                 your model classes
            views/                  your view templates
                  layouts/          your page layout templates
      components/
      config/                 configuration files
      db/                     your database migration files
      doc/                    generated docs
      lib/                    additional application code
      log/                    log files
      public/                 images, javascript, stylesheets
      script/                 rails ruby scripts
      test/                   your unit, functional, integration tests, and
fixtures
      tmp/
      vendor/                 plugins
      Rakefile
      README
```

Under the app/ directory, your controllers, models, and views are kept together in their separate subdirectories.

To generate a new model:

```
$ ruby script/generate model mymodel
```

Which produces additional files:

```
myproject/
      app/
            models/
                  mymodel.rb
      test/
            unit/
                  mymodel_test.rb
            fixtures/
                  mymodels.yml
      db/
            migrate/
                  001_create_mymodels.rb
```

First of all, if you dont want to write any templates, especially when first starting out a project, you can utilize the Rails built-in "scaffolding" generator, which will make everything you need (controller and views) on the fly from a Model. Scaffolding is widely hailed as a great thing about Rails. But its important to keep in mind that its really just to be used temporarily until you write your real code and templates. Scaffolding is not intended to be used in your final application.

To generate a new controller:

```
$ ruby script/generate controller mycontr
```

Which produces additional files:

```
myproject/
      app/
            controllers/
                  mycontr_controller.rb
            helpers/
                  mycontr_helper.rb
            views/
                  mycontr/
      test/
            functional/
                  mycontr_controller_test.rb
```

Most of the files are stubs, containing only basic hints. You proceed to edit the files and write code to build your application.

Django

To start a Django project:

```
$ django-admin.py startproject myproject
```

After generating a new project, the directory contains the following files:

```
myproject/
      __init__.py                  (0 bytes)
      manage.py                    django script for this project
      settings.py                  django settings
      urls.py                      django URLconf (dispatch controller)
```

Then create an app:

```
$ python manage.py startapp myapp
```

Which produces the following files:

```
myproject/
      myapp/
            __init__.py            (0 bytes)
            models.py              your models
            views.py               your views
```

The __init__.py files are required by Python to recognize the directory as a valid module (library). Depending how you want to set things up, you might add a templates/ directory under myapp/, or you could put your templates in another place altogether.

Thus, generally the models, views, and templates are kept together under the app. This allows apps to be modular and pluggable, and transferable to other Django installations.

Opinion

The Rails project directory makes it very clear that there's a place for everything, and everything in its place. Files are organized by type-- views, controllers, models, etc. which means sections of your site could be spread across various directories. If you have a lot of files and want to modularize the code, you can use subdirectories. Still, a set of related files (model, controller, templates) are spread about in different directories under app/ which seems upside down to me. But the organization is clean, and with Rails' file naming conventions, it's probably not much of a problem (no doubt this explains why TextMate on the Mac is a preferred editor, as it includes an integrated file explorer).

In Django, the default directories are sparse. You get the minimum you need to get started. No more, no less.

The organization of Django's apps seems more intuitive to me than spreading related files across multiple directories, and seems to lend itself better to application level modularity. (On the other hand, Rails has a much stronger plug-in capability, and a growing repository of reusable Plugins that Django sorely lacks).

Installation / Directory Structure	**Rails**	**Django**
Installation	easy	easy
Directories	strict, thorough, by function	flexible, sparse, by app
MY RATING: (1=worst, 5=best)	**4**	**5**

6. Database and Models

A lot of complexity is encapsulated by database abstraction and mapping to application objects. Both Rails and Django share robust support, including:

- Both frameworks use an ORM model (object relational mapping), which means there's an mapping between database tables and corresponding object classes. Rails' ORM is called *ActiveRecord*. Django's is named *Model*. In other words, both frameworks provide an API customized to your database "for free".
- Both have you declare the field names, types and attributes in the class definitions. You can set constraints on each field (e.g. max number of characters), automatic values (e.g. creation date, update date), and relations to other tables (foreign keys) for one-to-many and many-to-many relations.
- Both frameworks support "lazy access" which means the database is queried at the last possible moment, avoiding redundant operations and improving performance. Both also support transactions (atomic database operations), provided the underlying database supports.
- Both let you define "hooks", functions that are attached to specific operations without having to change the model code itself, such as before create, after create, before save, after save, etc. In Rails they're called "*observers*". In Django they're called "*signals*".
- Support the leading database servers, including MySQL, PostgreSQL, and SQLite. Rails also supports Oracle and MSSQL (in development for Django). I will not address the lower level database abstraction and driver layers, although there appears plenty of room for discussion in that area too.

Rails

Rails includes a powerful migration tool that lets you define your database schema in a series of files called "migrations." Migration files, named something like *db/001_create_mymodel.rb, db/002_adding_a_column.rb*, etc, define your tables' field names and types (text, integer, date, etc). Migrations are subclass of *ActiveRecord::Migration*

When you run the migrate command (*rake db:migrate*) your database is (re)structured accordingly, processing any new scripts since the last time. Migrate keeps track of the current migration level, and lets you roll back to an earlier schema.

Next, you define the model class for each data table, containing the business logic associated with that data (ie. *app/models/mymodel.rb*). The model may define validation rules, such as *required*, *unique*, or *length*, and used in user input forms. The model also defines any relations with other tables (such as one-to-many relationships). You would also define any model-specific methods helpful for accessing and updating data.

The Rails naming "magic" is really quite funky so I'll give an example here. Let's say you create a model "line_item" with the command

```
$ ruby script/generate model line_item
```

This may create a file db/migrate/006_create_line_items.rb which defines a `class CreateLineItems < ActiveRecord::Migration`. It also creates a file app/models/line_item.rb (note singular) which defines a `class LineItem < ActiveRecord::Base` . Classes are singular, Tables are plural.

The model inspects at the database columns (produced by the migration) to determine the method names available to the model object.

Rails also provides a tool for creating "fixtures", pre-defined data sets used (and re-used) in testing. Fixture data is specified in a simple language called YAML (which is easier to read and write than XML).

Rails supports model inheritance through a fairly simple mechanism of a single table with a collection of all columns from all child classes. This can lead to wide tables with many empty fields.

Finally, Rails has built in accommodation for 3 databases-- development, testing, and production. This is very handy and shows the framework is well footed in the real world.

Django

In Django, a model's table schema and methods are defined in a single models.py file. You add class variables which will become the database table columns.

Then use the command to generate the database table from the model, such as

```
$ python manage.py syncdb
```

The manage.py script actually can do a lot of things. Before syncdb, for example, you might want to try

```
$ python manage.py sql myapp
```

which will dump the SQL statements that syncdb will use to create the table, lets you doublecheck for errors in the model.

Django does not have a tool for migrations. It's under discussion (http://code.djangoproject.com/wiki/SchemaEvolution) but doesn't appear to even be slated for the 1.0 release. We're told basic migrations are easy to do directly in SQL, and the "hard" stuff, well, even Rails can't do properly (ref: irc log 2/18/07)

Django is working on model inheritance for the 1.0 Release. The plan is to use "joins" of separate tables for each subclass, a more thorough approach than the Rails one, but also more complex to implement.

Django can easily integrate with legacy databases, using a utility that introspects an existing database and produces a corresponding models.py file.

Opinion

To summarize, in Rails you create the database table separately, and the Model class inspects the table column definitions to determine its attributes. Django is opposite, you define the data attributes in your Model class, and that is used to create and drive the database table. Rails does pluralization magic; Django is explicit.

The Rails approach makes it difficult (if not impossible) for it to integrate with legacy databases.

In my opinion, the Django approach to Models is cleaner. The Rails implementation is more mature and supportive of the whole development process.

The Rails migration tool is a powerful approach supporting agile development, and in my opinion, it's an significant deficit in Django. I also really like how Rails has integrated the notions of development, testing, and production databases into its configuration.

Conclusion: Both frameworks appear to provide a robust and complete ORM model. Django is weaker without features like migrations and model inheritance (and legacy support is not so important to me). Rails wins 4 to 3.

Database / Model	**Rails**	**Django**
Schema migration	“Migrations”	manual
ORM class	“ActiveRecord”	“Model”
Model attributes	introspection	delcarative
Relations (1:1, 1:N, N:M)	yes	yes
Pluralization	yes	no
Hooks	“observers”	“signals”
Transactions	yes	yes
Model inheritance	yes (single table)	under devel (joins)
Devel vs Testing vs Production	built-in	no
SQL servers	MySQL, SQLite, Postgres, MSSQL, Oracle, Db2, Sybase, more	MySQL, SQLite, PostgreSQL (MSSQL and Oracle in devel)
MY RATING: (1=worst, 5=best)	**4**	**3**

7. Routing and Controllers

URL Routing

URL routing is how the framework interprets an incoming request from the user's browser, and decides which function to direct it to. The router allows “friendly URLs”, that are human-readable and don't have “those scary ?x=something blah blah things” in the strings (as someone I know put it)!

Rails

In Rails the config/routes.rb file defines the URL mappings. For example,

```
map.connect ':controller/:action/:id'
```

will route any URL in this format to the corresponding controller class, action method, and pass along the id as an argument. Thus, http://www.mysite.com/products/detail/123 goes to the *ProductsController*, “*detail*” action, for *id = 123*.

But you can change the mapping with new rules as needed, perhaps even change the url layout of the site without out changing any of the link_to, layout, form tags, etc in the code or templates.

Rails provide a URL generator (url_for method) so you do not need to hardcode real URLs in your code or templates. It uses the rules defined in routes.rb, so that logic is maintained in a single central location.

Django

In Django, the *urls.py* file defines the URL mappings. It uses standard (although cryptic) regex (regular expression) syntax to parse the URL string and map it to your functions. For example,

```
(r'^(?P<object_id>\d+)/products/detail/$', 'store.products.view'),
```

will route any URL in format www.mysite.com/products/detail/NNN to the app “store”, view “products”, method “detail”, with argument id = NNN.

Note, you explicitly state the mapping from the URL strings to the corresponding class and method. The URL is completely decoupled from the structure of your app. Thus, for example, you could rename your app or classes without affecting the URLs the rest of the world uses to access your site. And you can provide multiple routes to the same functions, for example to support legacy URLs on a new site.

Django recently added a template tag to generate URLs based on the urls.py configuration.

Django uses an HttpRequest and HttpResponse objects to pass state through the system. HttpRequest contains metadata about the request (e.g post and session info). Each view is responsible for return an HttpResponse object (e.g includes the HTML body etc).

Opinion

Django's loosely coupled URL mapping and regex syntax is much more flexible and thus more powerful.

Unfortunately I was born missing the "regex gene"! For years I have tried to learn it, and failed pathetically. Sure, I can copy and make small changes from examples. But I do hesitate to put a key feature of my websites into my regex-challenged hands.

With regard to loose coupling in Rails, there is a workaround, by creating a dummy controller class which then makes the call to the actual one. Its really a kludge, but it I suppose it works.

Conclusion: Django's loosely coupled, regex based URL mapping wins against Rails' declarative mapping, 4 to 3.

URL Routing	Rails	Django
URL Mapping	centralized, declarative (routes.rb)	distributed, regex (urls.py)
Reverse Mapping	complete	limited
MY RATING: (1=worst, 5=best)	**3**	**4**

Controllers / Views

The controllers are the real meat of your application. Controllers define the action methods that handle specific requests from users. In Rails this is the Controller class; in Django its called Views. (And what Rails calls Views are equivalent to what Django calls *templates*).

Surprisingly, I don't have a lot to say here about controllers. Or else I'd have too much. That's because when you get into the controllers, you're really describing the programming language and framework's API, libraries and helper functions.

In both Rails and Django, the controller actions are usually pretty short, just a few lines of code. They grab some data via the Model class, perhaps do some processing, and package variables that are passed into the templates. They might also handle error conditions and exceptions (for example, in a shopping application, if the URL requests a product Id that doesn't exist in the database).

Rails *scaffolding* provides a quick and dirty automatic generation of the Controllers and Views for a model. These are intended as temporary placeholders while you incrementally build your app.

Django comes with built-in generic views. Not the same as Rail's scaffolding, these are handy for common actions such as list/detail interfaces, archive pages, and a set of view for creating, editing, and deleting objects. Of course, you still provide individual templates for any custom views.

In Django, you can augment views with "decorators", a Python shortcut construct for specifying meta-attributes and contstraints. For example, a view method preceeded with @login_required will use the integrated authentication system to verify the user is logged in or he'll be redirected to a login page. In Rails you can accomplish the same thing with "filters".

Conclusion: I rate both a 3, and any real differences are covered in the other criteria.

Controllers / Views	**Rails**	**Django**
MY RATING: (1=worst, 5=best)	**3**	**3**

8. Templates, Forms

Templates

Obviously both Rails and Django support display templates, as this is a cornerstone of the MVC architecture separating content from presentation. Templates define the layout and formatting of data, for example, into HTML that is sent back to the user's browser in an HTTP response.

In general, templates are intended to be created (or at least edited) by non-programmer designers, an important consideration on team projects. Some debate has ensured regarding just how much programming features should even be permitted in a templating language.

Rails templates let you use the full power of the Ruby language, but you are "strongly encouraged" to use discretion and not put business logic into your templates (move it into the Controllers instead).

Django templates have their own little language that is intentionally limited to simple conditional and iteration control structures, and oriented toward non-programmers (i.e. designers).

Both frameworks permit page layout templates for sharing layouts among views, and specific templates for individual content. In Rails you use "layouts" (parent page), "partials" (included sub-page), and "collections" (an iterated partial). You can write "helper" functions to extend your tags.

Similarly in Django a template can "extend" a parent template (comparable to Rails' layouts but can be multi-level), "inclusions" of sub-pages, and write custom filters and custom tags. In both frameworks you can pass variables to the various template components.

Both frameworks permit you to replace the default templating system with third party alternatives. If you're using Rails but prefer Django's templating language for your designers, for example, someone has written a plugin, named "Liquid" which does just that.

Opinion

It's nice to have the full power of Ruby inside the templates. But I see the point about it being dangerous. And if you're really doing that, the logic should be moved into the controller. Otherwise they both have very good templating features. So I call it even, 3-3.

Templates	Rails	Django
Language	Ruby	Django template language
Parent layouts	“layouts”	“extends” directive (bottom up)
Include templates (children)	“partials”	inclusion tags
Extensions	“helpers”	custom filters and tags
Can use alternatives?	yes	yes

Templates	Rails	Django
MY RATING: (1=worst, 5=best)	**3**	**3**

Forms

Basically the role of a framework with regard to forms includes

- Generate HTML <label> and <input> tags based on Model class attributes
- Produce those fields in an empty <form> for creating new objects
- Produce those fields in a <form> with values filled in for updating an existing object
- Validate values posted by the form based on the Model's validation rules
- Return to the form, with error messages if any fields do not validate

The logic and syntax may vary somewhat, but both Rails and Django have very good forms support.

Django is presently in the process of rewriting its forms classes to be more flexible. It is available for download from the repository.

Rails includes support for forms that populate multiple Models. Rails also introduces the notion of "form_helpers" that help you standardize the look, layout and behavior of difference forms across your website.

Conclusion: I'll rate forms in Rails and Django equally, 3-3. I can see that Rails' is powerful, and we'll give Django's newforms the benefit of the doubt.

Forms	Rails	Django
MY RATING: (1=worst, 5=best)	**3**	**3**

9. Data and User Administration

Data Administration

Django

One of the things that Django is "famous" for is its built in administrative features. Given a Model, the framework can automatically generate a nicely designed set of admin pages to create, retrieve, update, and delete (CRUD) items in the model. It can also search, filter, and sort the lists. All the models which you've exposed to the Admin class appear on the admin/ home page. Unlike Rails' scaffolding, the admin is polished enough that it could be given to end users in a production website (such users being content editors, not visitors).

To enable Admin features for any model, simply add "class Meta:Admin:" to the Model class and refresh the page. Sweet!

However, as great as it is, I am told the code is very difficult to modify. So either you like and use it, or you're better off writing your own from scratch. This situation will soon change, as the newforms forms library is being integrated into a new Admin app, and when they're released you'll have the full glory of the slick admin interface, plus the easy custom-ability of newform. That's the promise anyway.

The Djang admin include basic permissions to add, edit, delete items. It is rudimentary but may suffice in many cases. It is limited to class granularity (operations on specific object types) not attributes of those objects. For example you cannot grant permission for a user to edit only items that he had submitted.

Rails

Rails has nothing like Django's admin GUI, in its core distribution.

However, Rails does have a number of non-core plug-ins that can automatically generate an admin interface. One is called Streamlined, and is actively under development. There is a webcast of the plugin and it looks quite slick (although I haven't tried it yet myself). (Another Rails admin plug-in called auto-admin is modeled after the Django one).

Otherwise, you are expected to write your own, as you would other Controllers + Views. Easy enough, if you don't mind.

Opinion

I should probably give more credit to the Django built-in Admin tool, but its not very customizable (until the promised revision). Rails is catching up.

One clever blogger reports that on his Rails projects, when he wants an admin interface, he uses Django to administer his Rails models! (A testament to Django's ability to interface with outside databases, by the way).

I've lumped in Permissions because its not developed enough in either framework to stand on its own.

Conclusion: Both framework's are working to improve their admin interfaces. Rating 2-2

Data Administration	Rails	Django
Admin GUI	via plug-ins, under development	Built-in admin, newforms version under development
Permissions	rudimentary, do it yourself	rudimentary, do it yourself
MY RATING: (1=worst, 5=best)	**2**	**2**

User Administration

Authentication

The Django Admin app also has basic user management, including administering users and groups, registration, and authentication (login).

Rails has stated they do not intend to include authentication in their core product. Instead there are plugins, including "act_as_authenticated", and several derivatives of this, which provide varying degrees of user administration and permissions. You then integrate this into your application.

In an enterprise environment, including corporate intranets, authentication must often be decoupled from the system to utilize single sign-on systems or directory services like LDAP. If this is important for your application, we'll leave it up to you to investigate whether the framework offers what you need (via user contributions, no doubt).

Registration

A corollary of this is registration, which refers to signing up new users. Most systems that have user accounts need a way for users to register themselves. And perhaps generates an email requiring users to activate their account (and thus validate the request). Both frameworks provide a basic level of support for registration, via user contributions (plugins, apps).

Sessions

Where there are users there must be sessions. Sessions enable continuity between HTML pages, which are inherently state-less. With sessions, the framework can know when a subsequent request from your browser is still you and not someone else (e.g. for showing error messages in forms, or for tracking items in a shopping cart).

Both Rails and Django have support for sessions and messaging.

For anonymous users, messages are passed between pages using an object called the "flash" in Rails. In Django you can pass messages via the "session" framework.

Neither Rails nor Django does automated housecleaning of expired sessions. You will need to setup a *cron* job or other script to periodically purge old sessions from the database.

Opinion

Both frameworks have a long way to go towards providing mature and robust user administration and authentication tools. They're nowhere near what I'm used to in the Xaraya web application framework. Yes, I understand a "development framework" is not an "application framework", but that's no excuse (we're in the Opinion section here!).

In general, other than some basic tools and components, you pretty much have to roll your own, especially, for example, if you need to build an advanced multi-user application with various groups and permission levels. I would love to see some seriously useful admin, user management, registration, authentication, and permissions tools to build upon.

And to be honest, I really need to play with both Rails and Django more, including any plug-ins and apps, to confirm or refute my concerns.

Conclusion: I rate both the same low score of "2".

Postscript: I recently discovered a Rails generator project called Goldberg that looks promising.

User Administration	Rails	Django
Authentication	as_authenticated plugin and its derivatives	Built-in admin
Registration	do it yourself	do it yourself
Sessions	yes	yes
User messages	“flash”	yes
MY RATING: (1=worst, 5=best)	**2**	**2**

10. AJAX

To quote the "Agile " book, AJAX "once stood for Asynchronous JavaScript and XML but now just means Making Browsers Suck Less." In general, AJAX refers to updating parts of the page without having the refresh the whole page. And thus it makes browser apps feel more like desktop ones.

Rails and Django have very different philosophies about the roles of a framework with regard to AJAX.

There are dozens of AJAX toolkits (libraries) available, both commercial and open source. One of the most important functions of these is to handle the idiosyncratic differences between browsers, as each browser implements and supports JavaScript differently. In addition, toolkits provide useful and fun things that you can add to your pages, like drag and drop, visual effects, etc.

Rails

Rails has fully embraced AJAX. And, in fact, it has standardized on a specific toolkits (prototype and Scrip.aculo.us) and encapsulated them so you can add AJAX effects to your pages using the Ruby and Rails code, without every having to touch JavaScript.

This is accomplished through a different "partial" template file format called .rjs. When you include an rjs template, it injects the appropriate JavaScript code into your page. You pass it variables, for example, that are passed to the Scriptaculous libraries. This is considered "server-side JavaScript" because you are programing your web app, as usual, on the server side. Although the JS does run on the user's browser, you're not necessarily thinking about that.

Django

AJAX has been hotly debated among the Django developers but their direction seems clear. Django does not intend to select or standardize on any particular toolkit. Instead, they will provide a library in the framework for "serializing" objects so they can be passed between your Django app on the server and JavaScript in the user's browser. There are standards for this, such as JSON - JavaScript Object Notation, which will be supported.

In this way, you are free to choose any AJAX toolkit you want, and integrate it with your Django application. (For example, the Dojo toolkit is well regarded).

The Django developers have publicly stated that like HTML and CSS, they regard Javascript a basic technology that should be mastered by any web developer. After all you don't ask the framework to write your stylesheets, do you? There are blog tutorials out there showing how easy it is to do AJAX with Django.

Opinion

While I understand the attitude of Django about AJAX, its not what I want right now. It may be what I need at some point down the road, but if I can get 90% of what I need wrapped up in a nice little package without having to program in Javascript, I'll take it. Then if I outgrow that wrapper, so be it, I'll research other toolkits, learn to use JSON, and so on.

Thus, I really appreciate what the Rails team has done with AJAX. Anything to get me quicker to my goal of a well-oiled, quality web site, the better. The fact they've established a strong foundation with the rjs template idea, all the better.

Django's serialization is under development.

Conclusion: I rate Rails a 4 and Django a 3 (I'm being generous to Django taking on faith that it's not so hard to roll your own AJAX with what they provide).

AJAX	Rails	Django
Support for AJAX	integrated prototype and Scriptaculous; rjs templates	object serialization
MY RATING: (1=worst, 5=best)	**4**	**3**

11. Other Features

The following features are also important considerations when evaluating a web development framework. However I have decided not to put a rating on them, either because there is not much at issue, or because there isn't much the framework really does, the features are premature, or its not a high priority for my projects. Your mileage may vary.

RESTful

RESTful is a clever approach to writing web interaction, using the actions built into the HTTP protocol, like GET and POST, as verbs on your application objects. For example a HTTP request (URL string) to display a product page is requesting a GET; and an HTTP request to save an order form is requesting a POST of that order. A single URL can be used for both of these, such as http://www.mysite.com/product/123 -- when its a GET you're getting the product Id 123, the same URL as a POST might be updating that product record.

- Wikipedia on REST: http://en.wikipedia.org/wiki/REST
- A Rails blogger comments on REST

There is nothing in either framework preventing RESTful programming, it's a methodology. However, Rails has added some library functions to facilitate coding, a scaffold generator for making skeletal RESTful controllers, and has begun to evangelize REST as part of the Rails development culture.

Conclusion: A nod to Rails for leading in that direction.

RSS/Atom/XML

RSS and Atom feeds are XML formatted data. Both frameworks are fully capable of supporting data requests in these formats. Python and Ruby include full support for XML structured documents.

Conclusion: It's needed, it's there.

Internationalization and Localization

Internationalization (enabling software for many languages, abbreviated "L18N") and Localization (implementing a specific language translation, abbreviated "L10N"), includes Unicode support, and string replacement.

Both Rails and Django support it, in different ways. Rails via a plug in; given the international popularity of the framework, no doubt it works. Django has native support; its admin app, for example, has already been translated into over a dozen languages.

Conclusion: This is not a high priority for me today, but no doubt it will be at some point, and I'm confident others are making it work.

Caching

Caching is the temporary storage of chunks of data so it can be retrieved much faster than the processing time needed to generate the data in the first place. Browsers have cache to retain pages and images that have been downloaded so you don't have to download them again.

On the server side, a framework cache will relieve the server from having the regenerate the same page chunks over and over again with repeated requests. This is most important and noticeable on high traffic sites.

Caching whole static pages (pages that don't change very often) is easy to automate. You still should be allowed to control when a page expires, however, to force a regeneration. On very dynamic pages, you can simply disable caching (you'd expect your shopping cart, for example, to be fresh each time it's reloaded).

Complexity comes when you try to optimize performance by caching static parts of a page and not caching the dynamic parts. You should be able to cache some page fragment and leave other fragments uncached.

Caching subsystems may also give you the option of caching on the server disk, in the database, or on a separate high performance memcache servers.

All this is fairly standard IT management, and both frameworks support caching.

Django adds support for caching binary objects as well. I don't know if anyone would bother to use it though. That level of fine tuning might be needed if you're approaching the traffic volumes of a Google or Amazon, but then you probably shouldn't be using either of these frameworks anyway.

Conclusion: HTML cache is necessary and supported in both.

Security

Security can refer to many things.

Whenever you have users submitting content, you are vulnerable to abuse. A common hack, cross-site scripting, is to try and insert tags like Javascript commands in content areas that allow HTML. A common solution is to strip any tags from user input, either before you save it to the database and/or when you display it. When you do need to allow tags in the content, be careful and selective with who input that text. This comes down to good programming practice, and is not very dependent on your framework itself.

Similarly, SQL injection is when a user inputs content that looks like SQL statement so when it's passed to the database trips up the database request.

Both the Django and Rails documentation offer advice on how to program properly to avoid these traps.

Security can also refer to permissions for accessing specific areas of the site, specific content, or specific operations on the content. For example, Visitors may view articles, but only Editors may submit them. This comes down to administration of users/groups and permissions. As mentioned earlier, neither framework offers very significant support for access controls, but do provide tools so you can build your own to your specifications.

Related to users, security also refers to authenticating users when the log in and/or hijacking a user's session. Again, Django has a rudimentary admin with basic authentication. Rails has is via separate plugins. Both support encrypted passwords and such. The rest is still up to you.

Hackers can try and break your site by entering invalid data in the URL, and if you don't trap the exception may result in an error page listing sensitive information and expose other nefarious ways into your system. Again, this is more about good programming practice and testing as you should be careful to trap exceptions created by invalid parameters to your functions.

Conclusion: At this time, it seems the best these frameworks can offer is some rudimentary security and a lot of documentation on best practices.

12. Documentation

Documentation serves not only its obvious purpose of teaching and reference, but also bolsters the community, helps recruit new users, promotes discussion and quality, and generally helps create momentum that leads to the continued success of open-source software (OSS) projects.

Documentation can be divided into the following areas:

- User manuals and tutorials
- API reference manual
- Books
- Community wiki and how-to's
- User blogs
- Video screencasts

In general, the first two "belong" to a project, and are hosted in the project's website.

User manuals are hard to get written, especially in a young project because they depend on developers who just want to write code not documentation. Manuals must address the needs of new users who are not familiar with a project as well as experienced ones. Thankfully, both Django and Rails have quite thorough on-line manuals. Both frameworks also have complete on-line API references (automatically generated from the code).

Hardcopy books, typically authored by one or more of the core developers and published by a commercial publisher, are a sign of maturity and stability of a project, and mainstream acceptance. Nowadays, PDF versions of books are often available as an alternative format from the publisher.

Rails has several books available, especially those from The Pragmatic Programmer . The popular "Agile Web Development with Rails" by Dave Thomas, and David Heinemeier Hansson is now in its second edition.

The Django Book is in draft form and is available on-line for review and comment. Some sections are still unwritten. It is being written for Django 1.0 release.

Both projects, having active communities, offer user-written wiki, how-to's and blogs that are easy to search via Google. Also add video screencasts to the list, increasingly popular they are especially effective for introductory demos of a feature or component.

Opinion

Fortunately, both Rails and Django have done a great job providing documentation. In fact, both projects offer more documentation than most other OSS projects I have worked with.

Django's book isn't finished but it's clearly preparing for Release 1.0. I have experienced network problems accessing the http://manuals.rubyonrails.com website.

One more thought (as good a place as any to bring this up)... I honestly enjoyed reading the "Agile" book and learned a lot. Ditto as I watched David Heinemeier Hansson keynote webcast from RailsConf'06 (http://www.scribemedia.org/2006/07/09/dhh). As much as DHH's personality can be annoying, he does give a great presentation and is a "thought leader." So with Rails, we're getting more than a nice framework, we're getting leadership on design methodology and an education...

Then again, the world is filled with lots of really smart inspiring people, and your framework needs to be ready to draw upon a world of resources.

Conclusion: Both projects offer very good documentation, although Rails is more complete. I give Rails a 4, Django 3.

Documentation	**Rails**	**Django**
User manuals and tutorials	very good	very good
API reference	good	good
Books	several	one, in draft
Community wiki and how-to's	good	good
User blogs	many	some
Screencasts	some	limited
MY RATING: (1=worst, 5=best)	**4**	**3**

13. User Extensions

The inner circle of developers on projects like Rails and Django are known to frequently debate where to draw the line between what should be in the core framework versus what should be an add-on component.

Frameworks rely on the user community to contribute pluggable extensions to the core framework. In Rails these are called *plug-ins*. In Django they are *apps*.

Rails

Rails has a well established Plug-ins capability. Plug-ins are easily installed and then appear to be a natural part of the core framework. Plug-ins may consist of code (classes), templates, stylesheets, and other files.

Plug-ins are installed using the "script/plugins" script (or its recent replacement "rapt", which retrieves a plug-ins code directly from a subversion repository over the Internet). Plugins get installed into your project's *vendor/plugins/* directory.

Presently Plugins cannot contain models, or provide modular “apps” like Django, although there are a couple of initiatives to add this (whether or not these play well or are just hacks remains to be seen). For example,

- Plug-ems: http://revolutiononrails.blogspot.com/2007/01/plugems-rails-as-first-class-citizens.html
- Rails Engines: http://rails-engines.org

A dedicated plug-ins section is maintained on the Rails site (http://wiki.rubyonrails.org/rails/pages/Plugins), and there's an online database of plugins at http://www.agilewebdevelopment.com/plugins I'm pointing this out because there's a lot of support for finding, using, installing, and writing plug-ins. As of Feb 15, 2007 there are 520 plug-in available (from about 55 repositories). The online database allows ratings to help weed out the junk from the jewels.

Django

In Django user contributions are like any other "app". You install them manually, which is trivially simple, nonetheless it's not just a single commandline command -- copy the code to your project's directory and the templates to your template directory; and then follow its documentation how to add it to your project.

For example ,some optional apps come with Django, include

- Flatpags app - for storing simple “flat” HTML content in the database, and handles the management via the admin interface

- Comments app – a simple flexible comments system (not yet documented)
- Redirects app – lets you store simple redirects in a database

Links to Django apps can be found at http://code.djangoproject.com/wiki/DjangoResources, a single user wiki page which also contains miscellaneous links to blog posts, tools, templates, code examples, tutorials, and so on. There is no coordinated effort to support user contributed apps, neither in terms of coding conventions for interoperability nor a centralized place to register find them.

Opinion

The quantity and quality of Rails Plug-ins is impressive. And Rails has made an effort to provide some infrastructure on its website for community development. Notably, the online database of plug-ins (with ratings) is not on rubyonrails.com, but a private consultants site, http://www.agilewebdevelopment.com/

As for Django, while the potential for user contributed apps is there, it has hardly been exploited. The limited infrastructure and quantity of Django apps is clearly disappointing.

Conclusion: For user extensions, I rate Rails 4 and Django 2

User Extensions	Rails	Django
Installation	command script, retrieves from a repository and installs in your project	manual: you get it, copy it, and use it
Official infrastructure	dedicated wiki page and detailed database of plugins	user wiki page, just a bullet list among all other resources
User contributions	hundreds available	a couple of dozen
MY RATING: (1=worst, 5=best)	**4**	**2**

14. Development Cycle

Debugging

While most of the discussion here so far has been about architecture design and API's for writing code, the bulk of a mortal programmers' time is spent troubleshooting and debugging. To the extent that a framework can help prevent bugs before they happen, all the better. But eventually you need to get down with it.

First, to state the obvious (I hope): To use any tool like a web development framework, you need to learn the programming language! (Seriously, too many people jump in before accepting that). Your debugging environment is primarily a function of the programming language.

Tools in your debugging arsenal include:

- Interactive command line console
- Framework generated error pages
- Dumping values from your program to a page
- Log files
- Breakpoints

Both Ruby and Python are interpretive languages, and their program are often referred to as *scripts*. Both languages offer an interactive shell-- a command line environment where you can run scripts and execute expressions for testing, debugging, and quick operations. The Ruby console is invoked with "*irb". The Python console is invoked with "*python*", or the more powerful "*ipython*".

When a web application encounters an error, the framework may present an error page with details of the error condition, what file it occurred in, the values of variables, a stack trace, HTTP header, POST variables, and so on. Debugging should be enabled in your development environment and disabled in your production one (as these error pages could contain sensitive details about your site).

Inside your code and templates you may need to dump the values of variables. Both framework's languages provide methods for converting variables to strings and including them in the HTTP response buffer (that is, your page). Rails has a "debug" helper method.

Log files are another invaluable tool for seeing what's going on (and going wrong) in your application. Both frameworks (optionally) generate log files, and you can add your own messages for tracing through problems.

Finally, you may need insert a "breakpoint" in your program, to stop execution and then examine for example the value of specific objects. Rails provides a built in breakpoint method, for use in a local development environment only. For Django you'd need to go to an IDE (as far as I know).

Some people like to work within an IDE (interactive development environment). The Rails folks say this is completely unnecessary but possible. A couple of popular ones that work with Python and Ruby are Eclipse and Komodo. (The Python plugin for Eclipse is Pydev).

Conclusion: It appears the debugging tools for both frameworks are mature and comparable, rated 3-3.

Testing

A really great way to preempt bugs is to *test*. Write code, then write tests that exercise the code, and validate the results. When you change things, you can re-run the tests to make sure things still work. Testing should not be an afterthought. (Taken to the extreme, test-driven development methodology encourages writing the test cases *first*, and then write the code to be tested.)

Testing also includes performance testing and profiling, for fine tuning your deployed application.

Robust testing tools are available in both the Ruby and Python programming environments.

Rails

Rails make full use of the Ruby language testing tools, and adds a nice testing framework within Rails. Each time you generate a new model and controller, Rails creates stub files in the project's *test/* subdirectory, ready and waiting for you to code them up.

Test scripts typically call the model and controller methods, and *assert* the results. Rails makes a clear distinction between "unit tests" for testing models (e.g. file *test/unit/mymodel_test.rb*), "functional tests" for testing controllers (e.g. file *test/functional/mycontr_controller_test.rb*), and "integration tests" for testing the work flow through your application. (Taken further, you can even develop a *domain-specific language* for application testing).

Test fixtures define a separate database specifically for testing. When you run a test, the database is re-created from the fixtures, ensuring that your tests are repeatable and consistent. Rails fixtures are written in YAML (.yml files), a simple language.

Django

With the standard Python *doctest* module, you run some tests of a method in the interactive console, and then paste the results from the console into the program script as a *docstring* comment. Then when you run doctest on that file, it re-plays the test and compares the results. Thus, the code, test cases, and expected results are all maintained in the same file.

You can also write separate test scripts, saved in *tests.py* file. Django's test runner looks for *doctest* and *tests.py* in your application directory.

An alternative is the standard Python *unittest* module, for which you write unit tests for your models.

Python's *test client* can be used to simulate the request/response interaction of your application with browsers.

Django does not have *test fixtures,* although they're said to be planned.

Opinion

None of the tools described above are replacements for browser automation testing frameworks, such as Twill and Selenium for testing of your web application independent of your specific application framework and language.

Programmers should write tests to test their own code. Other tests might be written by a separate QA person, to double check the code and perform application integration. In this case, maintaining test scripts in a separate test directory is better.

Much like documenting code with comments, writing tests is a necessary burden that requires professional discipline, because programmers normally just want to write code. The fact that Rails generators create test file stubs is like your father reminding you to do your homework, a not-so-subtle reminder of what you ought to be doing.

With regard to test fixtures, for smaller tests they seem fine. For more extensive testing with larger datasets, it seems it'd be burdensome to create and maintain long scripts and easier to just copy a previously created database to the test one.

Conclusion: While testing tools are largely available in both, Rails has more successfully integrated testing into the development cycle. I rate Rails 4, Django 3 for testing.

Deployment

While really not a function of the web development framework, deployment of your application to a live site is certainly an important part of the development cycle.

I will not go into the issues of setting up and administrating a website for Ruby or Python, nor the decisions related to Apache versus Lighttp web servers, etc. There are always issues and obviously the typical ones have been worked out. Many people will prefer to use a hosting service.

Scalability is also a consideration, to support high traffic volumes. Both Rails and Django are scalable to clusters of multiple servers.

Rails

Rails comes setup to easily switch between different configuration environments, namely, *development*, *testing*, and *production*. Each environment may have its own database, and its own configuration parameters (including logging, caching, debugging, and so on). You can even specify the version of Rails to use, so you can roll back to a previous version, for instance, after an upgrade.

Rails includes Capistrano, a Ruby-based utility for setting up a repeatable deployment scheme on remote servers. Capistrano assumes you store your code in a source code repository (such as subversion), and it assumes you are deploying to a Unix based server. Once setup, it's easy to update our live server from your (tested) development repository.

Configuring your web server is relatively tricky. *Mongrel* is a back-end Ruby web server. A recommended server configuration for Rails uses HTTP proxying to one or more *Mongrel* servers. Rails has problems with *fast_cgi*, and is not recommended.

There are hosting services dedicated to hosting Ruby on Rails applications. People have reported problems with some of them, so choose carefully. A recent review can be found at http://nubyonrails.topfunky.com/articles/2007/02/24/the-host-with-the-most

Django

Apache with *mod_python* currently is the preferred setup for using Django on a production server. (It can also run with *fast_cgi*). It all seems pretty standard as for any Python app, nothing special.

Opinion

Hosting a Django site appears easier, especially for smallish sites where HTTP proxying wouldn't be necessary. Otherwise, Django and Rails are comparable.

Rails has thought through the workflow of development-to-deployment more thoroughly and helps make it easier, like the different configuration environments and the Capistrano tool.

Conclusion: Rails gets a 4 for being more helpful, versus 3 for Django, even though Django may be easier for hosting small sites.

15. Conclusions

Tally the Ratings

Evaluation Criteria	Rails	Django
TECHNICAL		
Programming Language	4	4
Framework Concept	3	4
Directory Structure	4	5
Database / Model	4	3
URL Routing	3	4
Controllers / Views	3	3
Templates	3	3
Forms	3	3
Data Administration	2	2
User authentication, sessions	4	3
AJAX	4	3
RESTful	x	x
RSS/Atom/XML	x	x
Internationalization / Localization	x	x
Caching	x	x
Security	x	x
TECHNICAL SCORE	**3.18**	**3.27**
SUPPORT		
Project / Community	4	3
Documentation	4	3
User Extensions	4	2
Development / Debugging	3	3
Test Tools	3	3
Deployment Support	4	3
SUPPORT SCORE	**3.67**	**2.83**
OVERALL SCORE	**3.35**	**3.12**

Evaluation Results

To state the obvious, your priorities and ratings may vary greatly from ours, leading to completely different results. But let's analyze my results.

In the Technical categories, Django eeks ahead by just 0.09 point, with a score of 3.27 versus 3.18 for Rails.

In the Support categories, Rails shines, with its strong community and attention to the whole development cycle, rating 3.67 to Django at 2.83

And that puts Rails over the top for a final score of 3.35 against Django at 3.12, a 0.24 spread.

RAILS: The Winner!

DJANGO: A close second.

Of course, the story is far from over.

Rails is vulnerable. If Django builds a stronger community and support infrastructure, and succeeds in completing the features promised for its upcoming Release 1.0 and beyond, it could pull ahead as the superior choice.

That said, Rails can keep its lead by bolstering some of the technical areas where it lost points. Although some scores were more a matter of taste, we already see improvements coming in core and from community plug-ins which will enable Rails to maintain its leadership.

We'll be watching... :)

This all in fun. Clearly both frameworks are excellent products, offering improved productivity and higher quality web application development. The two camps are exchanging ideas, as cross-pollination is inevitable. And, as they say, a rising tide raises all ships.

Summary - Making Your Decision

First, you might want the programming language to be the deciding factor in choosing a web development framework. If you're already experienced in a programming language, and want to stick with it, then your decision is easy(ier).

If you are a *Pythonista*, then by all means, stick with Python. In that case you may still be doing a framework evaluation, as there are several competing ones in Python including Django, TurboGears and Pylons.

Similarly, if you are a *Rubyist* and love it, then stick with Ruby and use Rails, or consider one of the competing frameworks in Ruby.

If you are programming language agnostic, or don't mind learning a new one, then take the time to get to know the language syntax, communities and available resources. What are your friends, colleagues or employees using? That could be a factor too.

Second, do you like being an early adopter or are you more of a conventional buyer? While all MVC frameworks are relatively new and early, Django might appeal more to early adopters at its current stage.

Third, are you more comfortable relying on user contributions, or do you generally prefer to roll your own? Rails has a much larger body of plug-ins and other resources than Django.

And finally, do you like the efficiency of having things packaged together for you, or do you prefer the freedom of making your own choices? Rails is more "opinionated" and does more packaging. Django, in its *Pythonic* way, leaves more choices up to you.

Good luck, Have fun, and Build cool stuff!

SPECIAL THANKS to the Rails and Django developers for doing what they do!

About Parkerhill Technology Group

Jonathan S. Linowes is owner of Parkerhill Technology Group LLC (www.parkerhill.com) a web development firm based in New Hampshire USA, specializing in Web 2.0 applications development, product management and business planning. Contact us a info@parkerhill.com