

Archive for the "Django" category

Django vs Pylons

admin · Sunday, November 29th, 2009

This is an attempt to make a non-biased comparison between Django and Pylons. I did a brief comparison of the two at work since we needed a web framework for our new project.

We are making a web application that will serve as a frontend to a bank database. The application is going to allow bank employees to insert and track interbank loans and their repayments. The database is actually abstracted behind a layer built with Twisted, so the web application will only use XMLRPC calls to communicate with the database, hence no ORM.

As of the moment of writing, Django is in version 1.1 and Pylons is in version 0.9.7. Both of the frameworks are mature and tested in production in some major websites. Django has [Washington Post](#) and pylons has [Reddit](#).

The frameworks are quite different in philosophy, although both follow the same MVC paradigm. Django has more magic and less code, while pylons has more code and less magic. Pylons is essentially a bare-bones wrapper around the WSGI specification that uses 3rd party modules for templating, database interaction, routing and just about anything else, while Django is aimed towards rapid development of web applications and has everything packed inside of it - it's own template system, routing and ORM. This allows Django to establish high reusability for code between different projects. On the other hand, the developer is limited to one ORM and templating system.

Let's go point by point. These are not all the relevant features in a web framework by any means, but they should give the reader a general overview of the differences and the strengths of the frameworks in the most important areas.

Structure

Pylons reminds on Ruby on Rails here. It keeps all the controllers in one directory, all the models in another directory and all the templates in a third directory. Django is made to be more "pluggable". In Django you create apps that are self-contained, meaning they have their controller, models and possibly templates in one python package that you can easily pull out of the project and put it in another one. The "pluggability" of Django comes from the fact that you're using the same components in every project.

Pylons on the other hand is made to be more "swappable", meaning that you can easily switch a core functionality module, like the templating engine, with another one. This however, limits the reusability of your applications between projects.

I like Django's approach better because the files seem much better organized, but there is a slight disadvantage - the files have the same names. For example: if you have 4 apps, each of them contains a views.py file and it's a bit confusing when you got the mall open at once in an editor. But that's really nitpicking.

Documentation

Being a collection of 3rd party modules, Pylons' documentation is scattered all over the internet. You will need to read the SQLAlchemy docs for database, Mako templates' docs for templating, Routes' docs for routing, Paste's docs for general project tasks, Babel's docs for internationalization, etc. Django on the other hand has very extensive documentation on one place, and I must say it's much more comprehensive. There is one book for Pylons, but there are a dozen books for Django. This doesn't say much, except that Django is clearly more popular. This is a clear advantage to Django.

Community and Development

The Django community is huge compared to Pylon's. On [Stack Overflow](#) there are 3600 questions tagged "django" and only 100 tagged "pylons". Google trends shows similar results (around 15 times more for Django). I can't really talk about Pylon's community, but Django's is really friendly and helpful. Django also seems to be in much more active development than Pylons, but that's maybe because there is more work to be done. Pylons depends more on the development of the 3rd party modules it uses.

Learning Curve

This is somewhat dependent on the previous two points. A larger community, more code samples and better documentations works a

long way towards flattening the learning curve. On the other hand we should consider our previous experience with the two frameworks. If you already are familiar with SQLAlchemy and the WSGI spec, then Pylons requires less work to learn. Otherwise if you have experience with just python, then Django would be easier to grasp, only because of the availability of resources.

Coding required

Django, in principle, requires less code to do the same task. Less code means more magic and that is not always the best thing. Django was developed in a newspaper publishing environment where deliveries were really time-constrained and the authors created some very impressive tools to enable them to add website features quickly. For instance, Django comes with an automatic administration app that doesn't require you to write a single line of code. If you, however, strip Django of its ORM - much of this magic goes away.

Templates

Pylons the **Mako** templating language by default, but you can also use others, like **Jinja**. Jinja (as Jon pointed out) is based on Django templates and I find it much cleaner than Mako because it restricts you from running arbitrary python code inside template files (as does Django).

People tend to say that Mako templates are more powerful, but then PHP is also very powerful as a templating language and it always produces illegible tag soup. It's very tempting to start assigning and calculating values inside templates, especially when you're really tired.

Django templates and Jinja give you just enough power to get the job done and not a watt more :) They have only two special constructs - double braces for outputting values and brace followed by percent sign for invoking template tags. In Mako you have: "<%" for opening python blocks, "\${variable}" for variable printing, "%" for control structures, "<%!" for module-level blocks and "##" for comments.

Update: I just had a look at Jinja templates and it looks like they offer some interesting tools on top of Django templates, like simple expressions (e.g. `{{2+2}}`) inside the template tags. The only way to do some of the stuff that Jinja does out of the box with Django is by writing your own template tag. There is a nice post on why you should use Jinja, even with Django [here](#).

Anyway, you have the choice with Pylons.

Routing

Pylons uses the **Routes** library for routing. Routes is a clone of Ruby on Rails routing which is also inferior to Django's routing. Routes is implicit in a way that you define a route. For example `"{controller}/{action}"` routes to the 'action' method of the 'controller' class in your project. There is a possibility to use regular expressions but they come as additional parameters to the route declaration. Django's routes pair a regex to an explicitly specified view. Also they are much less verbose. As far as I can see (I may be wrong) Routes keeps all the routing declarations in a single file, while in Django you can chop off the first part of a URL and pass the rest to another url regex thus decoupling project apps.

Don't get me wrong here; both routing systems are capable, but Django's is just more natural and more pythonic.

ORM and Database

I don't have any experience with **SQLAlchemy**, but from what I read around it sounds like it's better than Django's ORM. It's the only option if you intend to use SQL Server anyway. Django was developed before SQLAlchemy was around so it uses it's own ORM, which is quite nice really. Coming from PHP, it's light years ahead of anything PHP has (take a look at **Propel**).

Since I can't say too much on this particular point, here's a few links that can fill you in:

<http://adam.gomaa.us/blog/2007/aug/1/five-things-i-hate-about-django/>

<http://reliablybroken.com/b/2008/06/choosing-sqlalchemy-over-django/>

<http://outatime.wordpress.com/2008/03/26/django-sqlalchemy/>

<http://jmoiron.net/blog/about-sqlalchemy-and-djangos-orm/>

Choices, Choices

In the end we went with Pylons. Our problem is not yet completely defined so we have to leave a lot of room for maneuvers. We don't know if the application will be just a single form or it will grow to replace the current desktop banking application. It's not going to have a database backend, no classic administration panel and no classic authentication. These are the areas where Django shines. If you leave them behind, the advantages that Django has fade out. Pylons however, with it's flexibility is a perfect choice for this kind of web applications.

On the other hand, if I had a standard website project I would go with Django all the way. It's really the best at rapid development of web applications and yes, the magic feels good. Sometimes you really don't care about what you are able to do **with** your framework, rather for what your framework can do **for** you. And Django can do an awful lot.

If you ask me, the biggest strength is the biggest weakness of both frameworks.

Pylons is a collection of 3rd party libraries. This means that you always get the latest and greatest of the Python world. But this also means that the community is much less compact and there is more manual work. Also, there is the problem of choice - you need to spend time and think about which component you'll use for each task. You don't want to end up with several projects all using different templating engines.

Django has the benefits of a huge, unified community, but it's weakness is that it's components are coupled and interdependent. [Here's a really good talk about the Django problems](#)

Posted in [Django](#), [Pylons](#), [Python](#) | [6 Comments »](#)

Maybe exceptions are not so awesome after all

admin · Saturday, August 15th, 2009

Maybe I got a bit rusty on my 1 month vacation, but today I ran into a simple problem that I spent 2 hours solving. I Am working on a website made in Django and I decided to make some changes to the model in one of the apps in the project. I currently have 3 apps - game, content and members. One of the classes of the content model imports a class from the game model, but I decided to remove that particular class.

After syncing the DB I noticed that the database tables for the content app are missing. This was strange, because no errors were reported. I tried to do a "python manage.py sqlall content" and the shell threw an error that it couldn't find the content app on my PYTHONPATH.

Error: App with label content could not be found. Are you sure your INSTALLED_APPS setting is correct?

I tried all kinds of different stuff until i found a mailing list that said that if you have import errors in some module, you cannot import it, which is to be expected, but the error messages that come up are all but informative. The message that my PYTHONPATH is incomplete in no way suggests that I have an import error.

What does this have to do with exceptions ?

Well, I'll talk about another example first: While I was writing a screen scraper, I used a lot of regular expressions. The regular expressions return a match object if they successfully make a match and None if they don't. If you try and call the .group() method on a Match object - it does the job, but calling it on a None object throws and AttributeError. I used to catch this exception in the upper layers of my program in order to avoid crashing the script on invalid input. This turned out to be a bad idea since all kinds of other stuff throws an AttributeError and makes the debugging a living hell.

I suspect that the same thing is going on in Django. Failure to import inside a module throws an ImportError which Django interprets as a missing module - only the module isn't missing - its a "submodule" that is missing. But the exceptions are the same, no matter on which level they happen.

Using exceptions correctly requires defining your own classes for every particular error that you may run into. This is a lot of extra code and need's to be weighed against the old-fashioned error codes.

A recursive django template tag

admin · Friday, June 5th, 2009

Here is my attempt to create a "silver bullet" tag for printing tree structures with the Django templating language. It's far from a silver bullet, tho, but it can do basic stuff:

It's a modification of the standard "for" tag and i have kept the counter, counter0, first and last variables, only this time they are not attributes to forloop, but rather to recurseloop. Check the docstring for more info.

For example, if you need to print comments that have other comments as replies, you would want the comments to appear one below the other, but the replies to be indented a bit. Let's say 20 pixels per level. So the code would be:

```
{% load file_that_contains_recurse_tag %}
{% recurse comment in comments children="replies" indent=(0,20) %}
    {{ comment.text }}
{% endrecurse %}
```

This tag will expect a list of comments (top-level) that have a property named 'replies' which contain other comments.

indent is an argument that will start with the float value of 0 and get increased by 20 on each depth level of the recursion. You can pass as many variables like indent as you like. They must be in the form

```
name=(float,float)
```

or strings will also work but must be enclosed in quotes

```
name=("string","string")
```

Caveat: You must not leave blank spaces between the equal signs when assigning children and additional incremented arguments. I will fix this later.

A second scenario is when you need the parent element to **contain** the children, like in unordered/ordered lists. In that case you can use the {% yield %} tag inside the recurse block. This tag will output the HTML between the recurse and endrecurse tags if there are any children in the current iteration item, or it will output nothing if there are no children.

```
{% recurse comment in comments children="replies" indent=(0,20) %}
    {{ comment.text }}
    {% yield %}
{% endrecurse %}
```

The yield tag (as for now) can **only** be used directly inside the recurse block, much like the {% else %} tag can only be used directly inside the if-endif block. This means that you can't make code like

```
{% recurse comment in comments children="replies" indent=(0,20) %}
    ({{ comment.text }})
    {% if cond %}
        {% yield %}
    {% endif %}
{% endrecurse %}
```

This will fail with an invalid block tag exception. You can check the docstring of the do_recurse function inside the code for more info. Also, you may want to check this code for errors since I just wrote it today, and haven't had much time to test it.

[Download and comment on any errors you find](#) :)

Python 2.6, Leopard 10.5.6 and psycopg 2.0.10

admin · Tuesday, May 12th, 2009

Psycopg2 is the [postgreSQL](#) adapter for Django. If you are trying to set it up on Mac OSX (using the supplied setup.py script) you may run into an error saying

```
File "setup.py", line 219, in finalize_options
NameError: global name 'w' is not defined
```

This is a simple thing to fix: open the setup.py file coming with psycopg2 and edit the line 219 like this:

```
except (Warning, w):
```

Remove the braces

```
except Warning, w:
```

In python 2.6 (at least) putting exceptions in brackets will catch multiple exceptions in this case Warnings and 'w's. The author clearly meant to get the Warning instance as 'w'. Also, do not forget to edit the setup.cfg file on line 28 and type in your pg_config path. Mine is

```
pg_config=/Library/PostgreSQL/8.3/bin/pg_config
```

and so should yours be if you used the default installer package for mac OS X. Oh, yeah, you need to install postgres before installing psycopg2.

Posted in [Django](#), [Mac OS X](#), [Python](#) | [One Comment](#) »

Configuring Django to work with your OSX X (Leopard) apache

admin · Friday, February 27th, 2009

I hope that I finally got it right, since I can see the admin interface and the media files are being served by the same development server as the site. The machine is an Intel MacBook running OS X 10.5.6 and python 2.6.1 I suggest reading the official [Django documentation](#) on setting it with up mod_python first. I hope that this article can fill in the gaps. Remember to change the paths and names to the ones that you use.

Configure the virtual hosts

In this case 'mysite' is the name of the virtual host and 'my_site' is the name of the project, and server root directory. The server root was in my /Users/discodancer/Dev/my_site directory

```
<VirtualHost *:80>
  ServerAdmin jordanovskid@gmail.com
  DocumentRoot "/Users/discodancer/Dev/my_site"
  ServerName mysite
  ServerAlias mysite
  ErrorLog "/private/var/log/apache2/my_site-error_log" CustomLog "/private/var/log/apache2/my_site
-access_log" common
  <Directory "/Users/discodancer/Dev/my_site
">
    Options FollowSymLinks MultiViews Includes
    AllowOverride All
    Order allow,deny
    Allow from all
  </Directory>
  <Location "/">
    SetHandler mod_python SetEnv DJANGO_SETTINGS_MODULE my_site
.settings
    PythonHandler django.core.handlers.modpython
    PythonPath sys.path+['/Users/discodancer/Dev/']
  </Location>
```

```
# Do not use python interpreter for /media
<Location "/media">
  SetHandler none
</Location>
# Do not use python interpreter for images
<LocationMatch ".(jpg|gif|png)$">
  SetHandler None
</LocationMatch></VirtualHost>
```

Then, to allow serving of media files, you need to make a symlink from django's contrib/admin/media directory to your project. The apache user normally does not have privileges to the django installation, so you need to do this.

```
In -s /Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/django/
contrib/admin/media/Users/discodancer/Dev/my_site/media
```

(the path is too long, try not to paste the line breaks in your terminal :) Then make a file `apache_settings.py` in your project directory/server root and paste these lines in it:

```
import osos.environ['PYTHON_EGG_CACHE'] = '/Users/discodancer/Temp'
```

The path in my case is writable by the webserver (anyone for that matter). Finally add these 2 lines in the apache `httpd.conf` file. They will tell apache to load the settings from the file you just created.

```
PythonInterpreter my_sitePythonImport /Users/discodancer/Dev/my_site/apache_settings.py my_site
```

Restart the web server. I suppose you already know, but the apache `httpd.conf` file can be found in `/etc/apache2/httpd.conf` and the virtual hosts file can be found in `/etc/apache2/extra/httpd-vhosts.conf`. This should work :) at least it did for me. One more note: at the moment of writing there is no current MySQLdb module for python 2.6. I am using the one that works with python 2.5 and each time I import it it throws a warning that the sets module is deprecated. Just ignore this, it didn't cause any trouble to me. If someone can explain what it really means, i'd be grateful.

Posted in [Django](#) | [No Comments](#) »

Install mod_python on Mac OS X

admin · Thursday, February 26th, 2009

This is not my article, i copied it from [here](#) for safe keeping.

First, you'll need to [grab the source](#) to `mod_python`. I recommend version 3.3.1, which is what I've worked with. Then, you'll need to unpack it:

```
$ tar xvzf mod_python-3.3.1.tar
$ cd mod_python-3.3.1$ ./configure --with-apxs=/usr/sbin/apxs
```

At that point, the configuration script will spit out a lot of things that you shouldn't really care much about. Just make sure at the end it spits out a bunch of things about creating Makefiles. From there comes the normal sequence of events with most open source software:

```
$ make$ sudo make install
```

The last requires the `sudo` command because it installs a bunch of pieces in privileged areas. Never run as root; always use `sudo` for your administrative needs. Finally, you need to add the module to your `httpd.conf` file, which is located in the `/etc/apache2/` directory, after making a back-up of course.

```
$ cd /etc/apache2
$ sudo cp httpd.conf httpd.conf.orig$ sudo vi /etc/apache2/httpd.conf
```

Then, scroll down to where you'll find all the `LoadModule` commands, and add another line:

```
LoadModule python_module /usr/libexec/apache2/mod_python.so
```

Now all that's left is to reload Apache to make sure it loads the module for you:

```
$ sudo /usr/sbin/apachectl restart
```

At that point, you're ready to proceed. The best place to start is the [mod_python documentation](#), specifically the [section on testing](#).

Posted in [Django](#) | [2 Comments](#) »