Python Web Environment Set up

A web environment needs to be scalable when request load raises. It consists of a load balancer, a web server serving static contents, and an application server hosting applications written in Java, Python, or other languages.

# Newbernetes: kubectl the Kubernetes CLI - CoryODaniel - MediumLoad Balancers

F5 (<https://www.f5.com/services/resources/glossary/load-balancer>) is a hardware based load balancer, very expensive and versatile.

Software based solutions are noted in <https://www.loggly.com/blog/benchmarking-5-popular-load-balancers-nginx-haproxy-envoy-traefik-and-alb/>



Notably, Nginx and HAProxy are very popular. We choose Nginx because it also serves as a web server. ALB is the load balancer in AWS.

To set sticky-session policy, see <http://nginx.org/en/docs/http/load_balancing.html>

# Global Network icon Web servers

To serve static contents, such as pictures, javascripts, we need a dedicated web server. There are a few web servers mentioned here: <https://geekflare.com/open-source-web-servers/>, though the main player is apache web server for many years. We choose Nginx because it’s performing and multi-functional.

<https://www.tecmint.com/why-nginx-better-than-apache/>

<https://www.monitis.com/blog/6-best-practices-for-optimizing-your-nginx-performance/>

<https://www.cyberciti.biz/tips/linux-unix-bsd-nginx-webserver-security.html>

# WSGI Servers

WSGI servers handles the concurrency of web requests so we have less concern in coding. Since Python has GIL (<https://wiki.python.org/moin/GlobalInterpreterLock>), we have to deal with high concurrency at certain point. There are several ways to deal with this, but we decide to handle this concern outside of our code so the code is cleaner. With this approach, there are mainly 2 choices, gunicorn and uwsgi. A good reference is <http://xplordat.com/2020/02/16/a-flask-full-of-whiskey-wsgi/>. So there is ignorable difference between the two. We choose uwsgi because it has longer battle-testing time.

<https://iximiuz.com/en/posts/flask-gevent-tutorial/>

<https://fgimian.github.io/blog/2018/05/17/choosing-a-fast-python-api-framework/>

# Session Storage

We use Redis as the session storage.

# Application Servers

Application servers host application logic. In Java, we have Tomcat, Jetty, etc. In Python we have many options, for example,

<https://steelkiwi.com/blog/top-10-python-web-frameworks-to-learn/>

<https://hackernoon.com/top-10-python-web-frameworks-to-learn-in-2018-b2ebab969d1a>

<https://stackify.com/python-frameworks/>

<https://hackr.io/blog/python-frameworks>

Among these, Django, Flask and Tornado are most popular. Django is quite heavy weight and tightly coupled. So it’s a canned solution. So we skip this for light weight solutions. Though Tornado has class based handlers, its async nature makes it impossible to utilize a WSGI server listed above. So our option is Flask, but the downside is that its handlers are function based, not classed based as in Tornado.



Here are some references for this setup:

<https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-uswgi-and-nginx-on-ubuntu-18-04>

<https://hackersandslackers.com/deploy-flask-uwsgi-nginx/>

<https://uwsgi-docs.readthedocs.io/en/latest/WSGIquickstart.html>

<https://iximiuz.com/en/posts/flask-gevent-tutorial/>

<https://fgimian.github.io/blog/2018/05/17/choosing-a-fast-python-api-framework/>

<https://hackersandslackers.com/deploy-flask-uwsgi-nginx/>

<https://hackersandslackers.com/redis-py-python/>

Flask Tutorials:

<https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-ii-templates>

<https://www.tutorialspoint.com/flask/index.htm>

<https://www.gab.lc/articles/flask-nginx-uwsgi/>

<https://hackersandslackers.com/deploy-flask-uwsgi-nginx/>

<https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask>

Flask client session example is <https://www.tutorialspoint.com/flask/flask_sessions.htm>. This is not useful for us.

Flask server side session example is:

<https://hackersandslackers.com/managing-user-session-variables-with-flask-sessions-and-redis/>

<https://flask-session.readthedocs.io/en/latest/>

Flask error handler

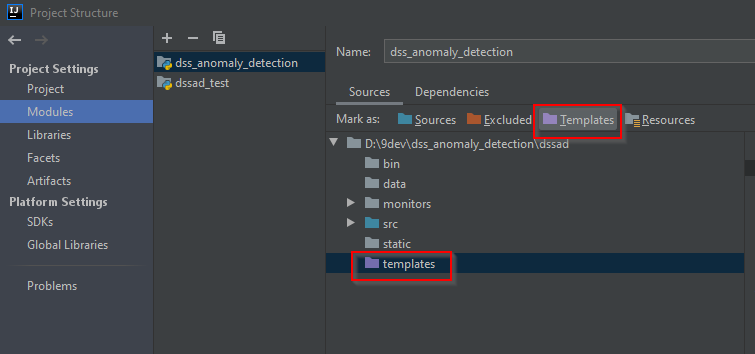
<https://towardsdatascience.com/how-to-do-rapid-prototyping-with-flask-uwsgi-nginx-and-docker-on-openshift-f0ef144033cb>

<https://www.toptal.com/flask/flask-production-recipes>

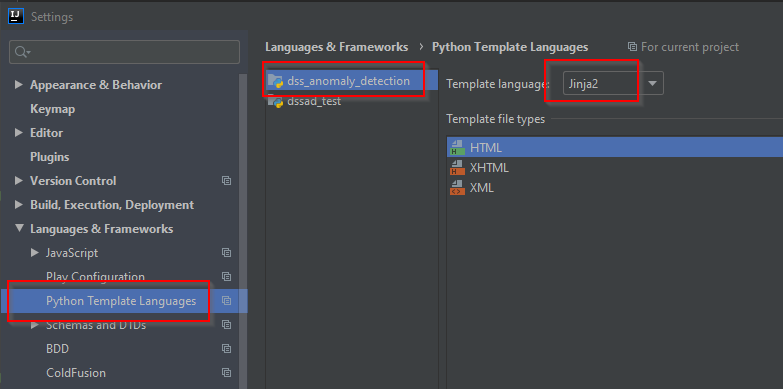
<https://www.e-tinkers.com/2018/08/how-to-properly-host-flask-application-with-nginx-and-guincorn/>

IDE template setup to avoid template file missing:

Go to project settings



Go to IDE settings:



# Process Monitoring and Notification

There is a process monitoring tool: supervisord

<http://supervisord.org/>

<http://www.inanzzz.com/index.php/post/bn7g/monitoring-processes-with-supervisor>

<https://linoxide.com/linux-how-to/supervisor-monitor-linux-servers-processes/>

<https://serverfault.com/questions/647357/running-and-monitoring-nginx-with-supervisord>

<https://github.com/mlazarov/supervisord-monitor>

We need a tool to monitor our running process and notify us in case of errors. Another possibility is the residence cloud, to see whether they have tools.

# Python Foundation Libraries

Favor **Dask** over Celery

<https://matthewrocklin.com/blog/work/2016/09/13/dask-and-celery>

Pandas can work with Dask.

Dask logging for communication between scheduler and workers and logging for workers. One way is dask dashboard.

<https://stackoverflow.com/questions/48560641/how-to-capture-dask-worker-console-log-in-a-file>

Another consideration is for all parallel computing, not specific to Dask.

Similar to JVM/Thread shared data, Linux shared memory seems the solution to avoid massive data issue:

<https://stackoverflow.com/questions/53351060/dask-shared-memory-in-parallel-model>

<https://statistics.berkeley.edu/computing/parallel>

<https://docs.python.org/3.8/library/multiprocessing.shared_memory.html>

[https://arrow.apache.org/docs/python/plasma.html#](https://arrow.apache.org/docs/python/plasma.html)

Favor **PyODBC** over others

Can work well with Pandas. I expect everyone writes SQL. For SQL based relational databases, we have tools to avoid SQL. For NoSQL databases, most of the time we don’t have these kinds of tools but they have SQL-like languages. So SQL is the common denominator across SQL and NoSQL. It’d better if we get used to it and are familiar with it all the time.

Another reason that I am not in favor of SQLAlchemy is that it pollutes the entity domain objects. It doesn’t work with NoSQL. Besides, it’s about 30% slower, like all ORMs.

<https://stackoverflow.com/questions/8812631/pymssql-versus-pyodbc-versus-adodbapi-versus>

last comment indicates that PyODBC is in a better position.

<https://stackoverflow.com/questions/29377756/pyodbc-doesnt-report-sql-server-error>

one of the ways to deal with missing error to be returned.

Google “pyodbc vs pymssql sored proc no error returned” for more hints.

Pandas:

It’s not likely that we are going to run SQLs in Pandas, though we can do it (package pandassql). The main concern is performance. In addition, Pandas can work with Dask and other tools for parallel processing.

If SQL is a must, consider in memory database sqlite:

<https://blog.thedataincubator.com/2018/05/sqlite-vs-pandas-performance-benchmarks/>

Error handling

Logging

Numba, faster python

<http://numba.pydata.org/>