

\* Necessary documentation for us to set up the service ourselves on EC2/Vagrant/Docker or similar system of your choosing. Please do not utilize services such as AppEngine or Heroku. Bonus if you do this using Ansible/Salt/Chef or other configuration management.

## Repo location

[https://github.com/chenjienan/url\\_lookup\\_service](https://github.com/chenjienan/url_lookup_service)

Currently the project is set to private, please email [chenjienan2009@gmail.com](mailto:chenjienan2009@gmail.com) if you cannot access the source code

## Project setup on remote cloud container

Setup IAM key and secret

Run application on cloud container

```
docker-compose -f docker-compose-prod.yml up -d --build
- run dockerfile
```

Recreate database

```
docker-compose -f docker-compose-prod.yml exec url_lookup python manage.py
recreate_db
- Drop table
- Create table according to the schema in model.py
```

Seed the Database

```
docker-compose -f docker-compose-prod.yml exec url_lookup python manage.py seed_db
- Add url: google.com and amazon.ca to the database
```

Run unit tests

```
docker-compose -f docker-compose-prod.yml exec url_lookup python manage.py test
- Currently I have 13 test cases
- Should cover the majority of the functionality in this app :)
```

Check code coverage

```
docker-compose -f docker-compose-prod.yml exec url_lookup python manage.py cov
- Code coverage is 66% (because of time)
```

## Project setup on local machine (already done on AWS EC2 instance 3.85.189.102)

Run application on local container

```
docker-compose up -d --build
```

Recreate database

```
docker-compose exec users python manage.py recreate_db
```

Seed the Database

```
docker-compose exec users python manage.py seed_db
```

Run unit tests

```
docker-compose exec users python manage.py test
```

Check code coverage

```
docker-compose exec users python manage.py cov
```

## Endpoint

Please feel free to test out the following endpoints

Endpoint	HTTP Method	CRUD Method	Result	Purpose
/urlinfo	GET	READ	Get url information	Requirement
/urls	GET	READ	Get all urls in the system	For testing only
/urls	POST	CREATE	Add url to the system	For testing only
/ping	GET	READ	Sanity check	For testing only

**Online deployed service (< \$1 CAD on AWS):**

<http://3.85.189.102/urls>

If you want to perform a post action, please use the following JSON

```
{ "url": "yahoo.com" }
```

And set format to JSON(application/json)

<http://3.85.189.102/ping>

<http://3.85.189.102/urlinfo/google.com:443/something.html%3Fq%3Dgo%2Blang>

## Dependencies:

Please check out the requirements.text file under `services\url_lookup\requirements.txt`

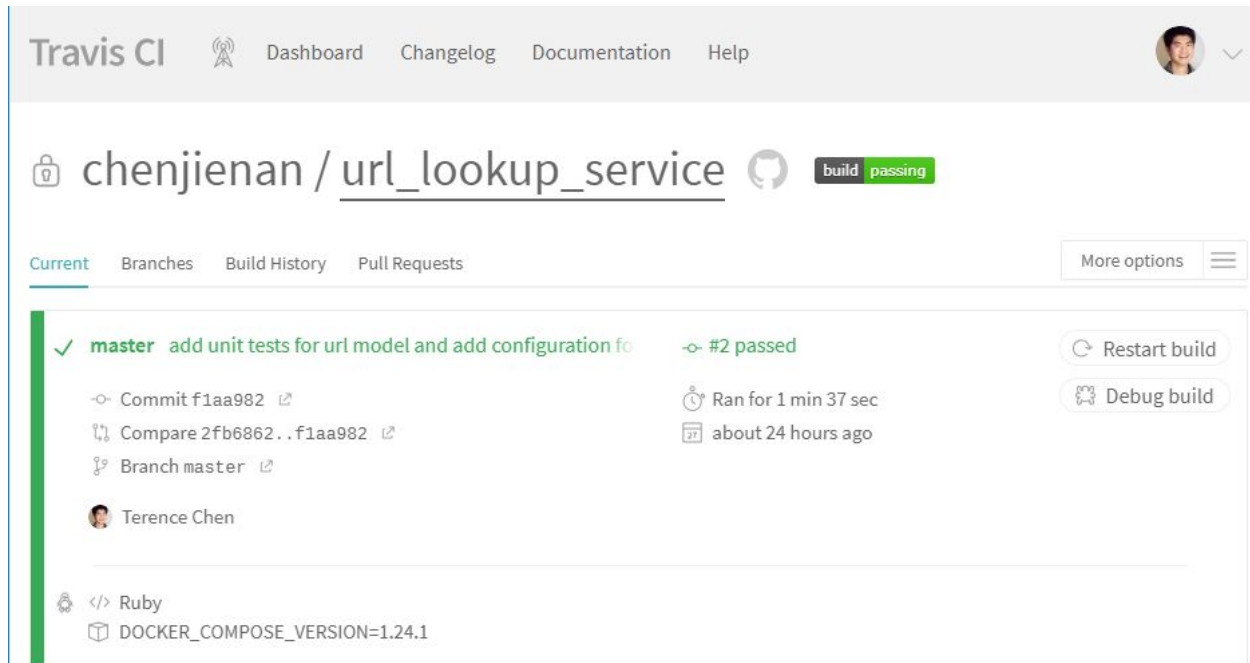
## Configuration management

I am using an online service TravisCI for continuous integration. The whole process is linked and automated with the Github repo. (Currently I am using the trial version for 100 builds only)

CI definition can be found under the root folder `/.travis.yml`

Please see the following screenshot for more information.

The CI info can be found on [https://travis-ci.com/chenjienan/url\\_lookup\\_service](https://travis-ci.com/chenjienan/url_lookup_service)



\* Documentation on your solution's result format/structure

### Database schema

id (PK, integer)

url (varchar(2048), unique)

active (boolean)

created\_date (DateTime)

**Result format** for [urlinfo/google.com:443/something.html?q=go+lang](http://3.85.189.102/urlinfo/google.com:443/something.html?q=go+lang)

// http://3.85.189.102/urlinfo/google.com:443/something.html?q=go+lang

```
{
  "status": "success",
  "url": "google.com:443/something.html?q=go+lang",
  "host": "google.com",
  "isMalware": "true"
}
```

According to this result, if **"isMalware" == "true"** that means the host name is in the system which is denoted as malicious, and the associated subpath will be marked as malicious as well.  
(Please correct me if this is wrong)

## Folder structure

Essentially, the application is running in three containers: **Flask**, **Postgres**, and **Nginx**. Based on the repo, you may see the structure of the project is quite intuitive.

On the other hand, we could use the development setting or the production setting.

\* Ability to handle an expected load of thousands of requests per second

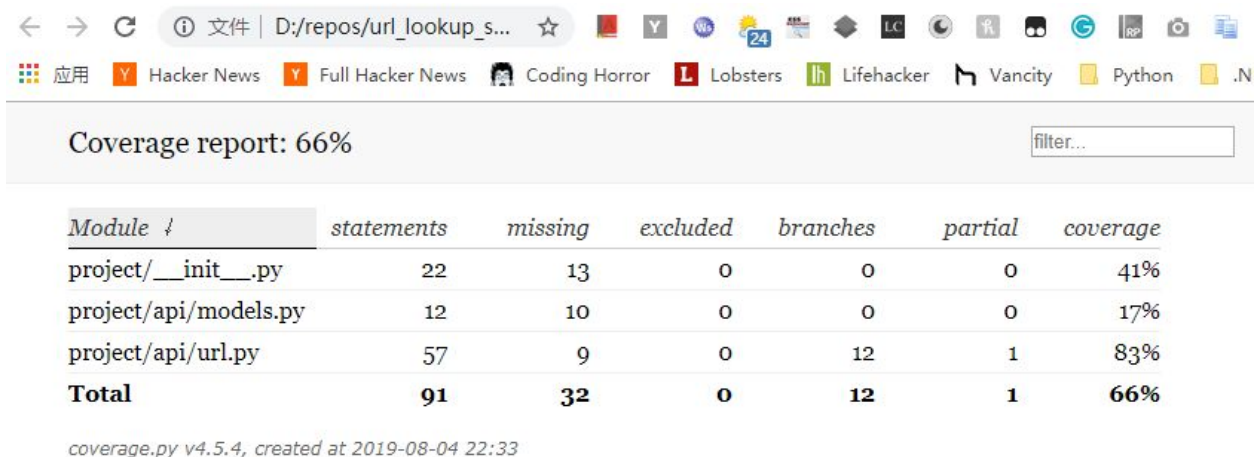
I am currently using PostgreSQL for data persistence (Please see `services/url_lookup/project/db`)

For a single SSD SQL service, the system should be able to handle QPS around 2000-3000. The database is located in a separate container. We can improve the performance if needed.

\* Unit tests. Integration tests would be great as well, but not required.

```
docker-compose exec users python manage.py test
```

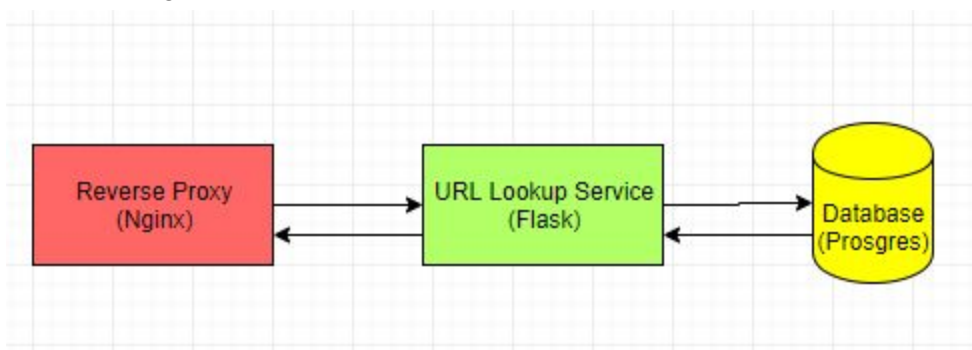
```
docker-compose exec users python manage.py cov
```



Module ↴	statements	missing	excluded	branches	partial	coverage
project/__init__.py	22	13	0	0	0	41%
project/api/models.py	12	10	0	0	0	17%
project/api/url.py	57	9	0	12	1	83%
<b>Total</b>	<b>91</b>	<b>32</b>	<b>0</b>	<b>12</b>	<b>1</b>	<b>66%</b>

*coverage.py v4.5.4, created at 2019-08-04 22:33*

Overall Design:



Give some thought to the following:

Note: Because of the time constraint, I won't be able to implement a highly scalable system according to these extra steps. However, the project lays out a good microservice infrastructure which is able to be deployed to a high performance cloud-based cluster.

\* The size of the URL list could grow infinitely, how might you scale this beyond the memory capacity of this VM?

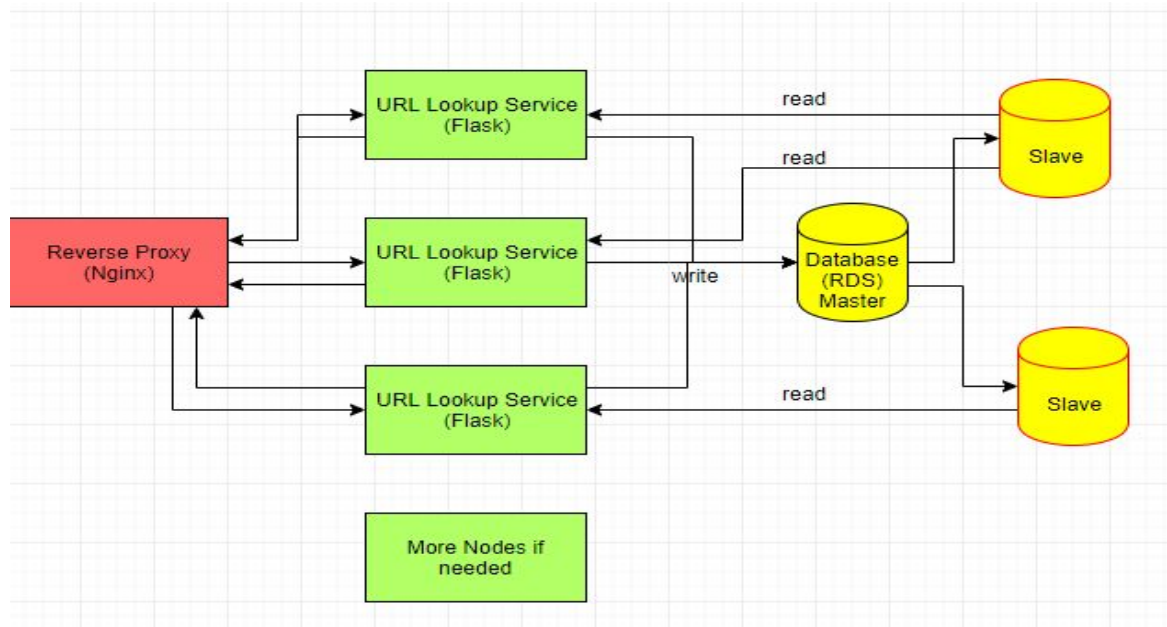
If the size of the URL list grows infinitely, we can set up a scalable cloud database for our data persistence layer (e.g. AWS RDS or Azure Database for PostgreSQL). The cloud-based database provides a highly available DB system which has tools for "scaling" a DB for more traffic (AWS Multi-AZ). On the other hand, using NoSQL may not be a bad idea (can be discussed if we have a chance)

In this case, data integrity is an issue. If the container crashes we may result in corrupted database. It's a good practise to centralize the data storage layer in our production cluster.

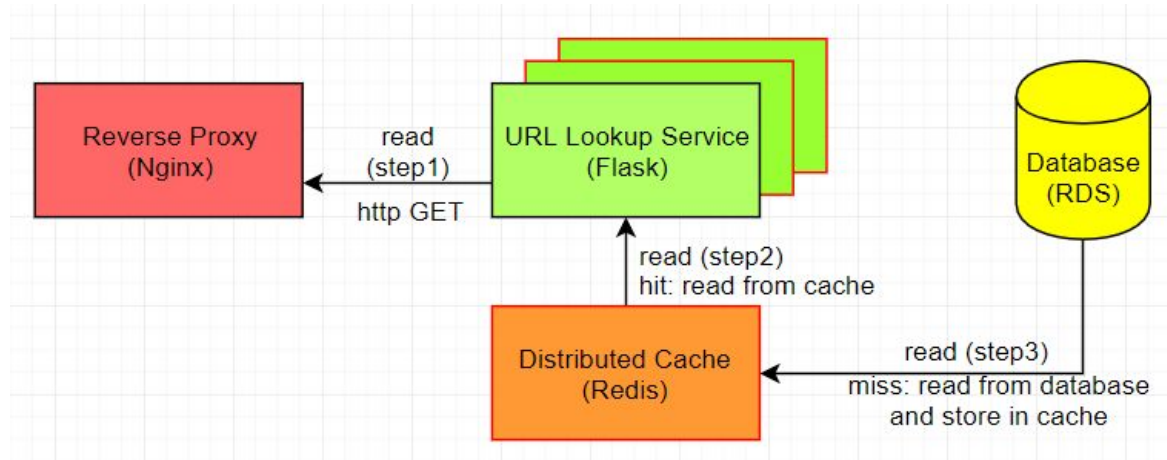
Last but not the least, we can save time and money using cloud-based database (RDS) rather than managing our own Postgres instance on a server somewhere

\* The number of requests may exceed the capacity of this VM, how might you solve that?

If the number of requests exceeds the capacity of the VM, we need to scale out the system by adding more nodes to handle the requests. In this case, we need to introduce the container orchestration (with Amazon ECS or Kubernetes) to a more scalable system. We can also add Amazon's Elastic Container Registry along with Elastic Load Balancing for load balancing and RDS for data persistence (as I mentioned in the answer above) and master-slave DB see below:



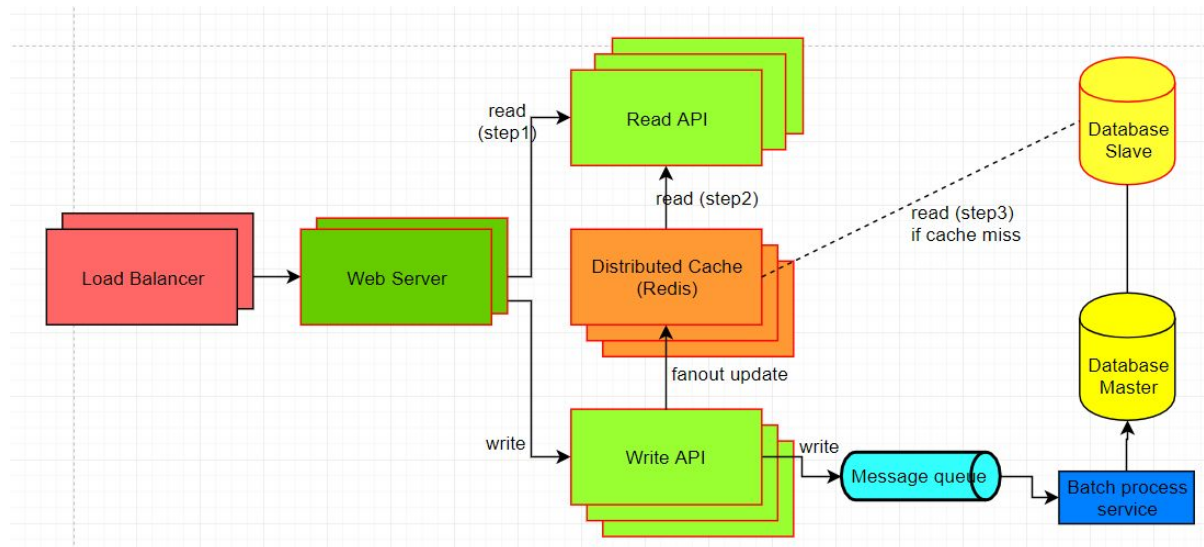
Another approach is to add a cache layer to the url lookup service (e.g. Distributed Redis). Therefore, we are no longer rely on the memory on a single node. However, we need to have a better strategy for cache invalidation and cache update. (see below design)



The read-through cache strategy may work better in this case. The cache sits in-line with the database, when there is a cache miss, it loads missing data from database, populates the cache and returns it to the application.

\* What are some strategies you might use to update the service with new URLs? Updates may be as much as 5 thousand URLs a day with updates arriving every 10 minutes. These updates will include insertions and deletions.

We should split the **read** function with **write** function with cache (see the design below)

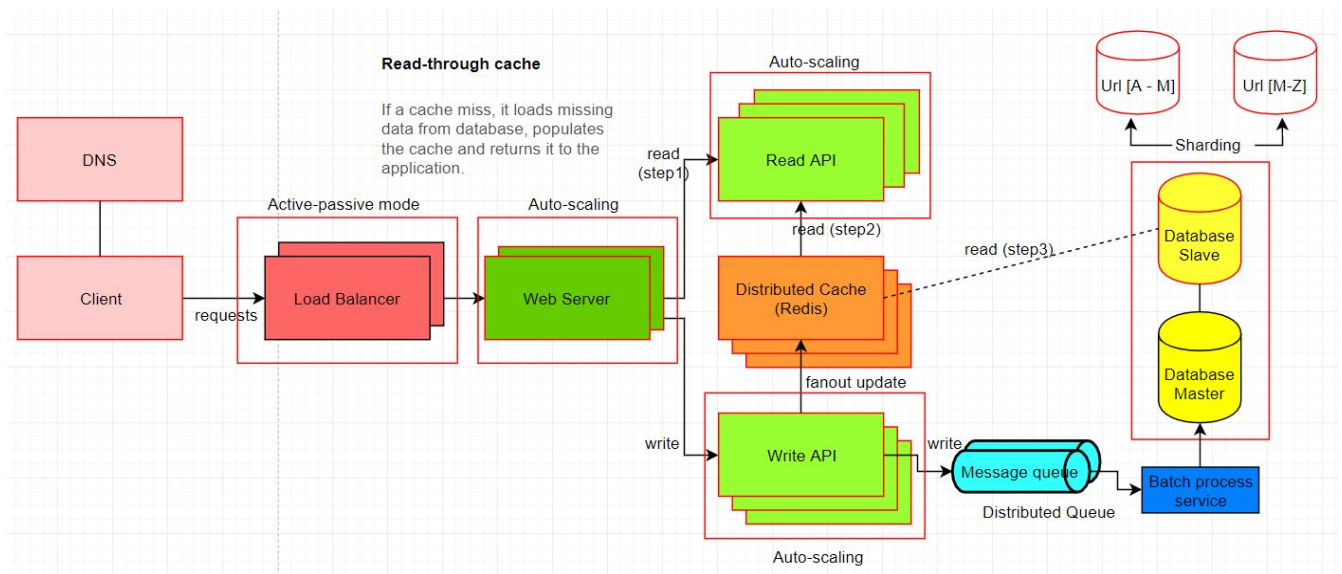


Based on the read/write ratio, we can setup auto-scale for read/write api nodes on the cloud. For a better read/write performance, it's good to use the **fan-out strategy**. For each single write action, we can update the cache first (so that end-users are able to retrieve the latest update in real time ideally), then use MQ (e.g. Kafka or AWS Simple Queue Service) to batch process a relatively large volume of database update to avoid heavy traffic on the database side.

\* How would you design the system to tolerate component failure such that you can achieve 100% uptime.

Based on the microservice cloud-based strategies we have as I mentioned above. We should have a good HA result. As suggested in ECS (container orchestration), we can have the following wish-list:

- **Health checks:** verify when a task is ready to accept traffic (ALB)
- **Zero-downtime deployments:** deployments do not disrupt the users (ALB)
- **High availability:** containers are evenly distributed across Availability Zones (ECS)
- **Auto-scaling:** scaling resources up or down automatically based on fluctuations in traffic patterns or metrics (e.g. CPU usage) (ECS)



We can refine the system in a more granular level. Following the availability pattern, we can use the active-passive fail-over heartbeat in the load balancer (with health checks), if the heartbeat is interrupted the passive server takes over the active's IP address and resume service.

In the database side, we continue using the master-slave replication and separate the reads and writes in a tree-like fashion. If the master is offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.

In reality, to achieve 100% is possible. We may need to eliminate the single points of failure with reliable crossover and better detection of failure when they occur.

Friendly reminder: the availability is often quantified by uptime as a percentage of time the service is available and it's generally measured in number of 9s. To achieve more digits of 9 we may need to add complexity and invest more money into the system.