SE−227

Handson #3

December 16, 2019

1 Part 1: Website Deployment

1.1 Dockerize

deploy components with docker-compose

Since almost all components of this website were released and distributed via the pre-built docker images, we can easily creates an docker-composed system via a simple yaml configuration files.

1.2 Dependencies

let them wake up sequentially

It is obvious that some components of this system dynamically depends on the others. For example, without the MongoDB system, the counterpart back-end system will refuse to boot, too.

So in the docker-compose.yml files, those MongoDB sub-services are declared with an extra property: the healthy checking part. Under this part we can use some simple bash code pieces to check if the system has been running correctly. Under this case, it's "db.stats().ok".

On the other hand, on those depending-on-other services, we just stab one "depend_on" field into their configuration phases.

Notice that we didn't declare any prerequisites for the fore-end service. The main reason is that there's actually no much cost for a fore-end web service to retry an internal request. And it won't refuse to start without the back-end. It will just wakes up, without any actual data and functions, and begin to run normally with a simple refresh.

However, without the MongoDB, the SpringBoot based back-end component will even not start. It didn't have a simple and usable retry system to recover from this situation. So, ensuring their wake-up after the dependents is very important.

1.3 Named Services

how they communicate with each other

It's obvious that in the internal implementation of each functional module, there are no hard-encoded IP addresses, but simple and meaningful names on it. That's exactly the service name we give them in the docker-compose configuration file. So, that's not a problem, too.

1.4 Port Mapping

mapping port for the frontend service

Don't forget to explicitly declare the port of the frontend service's port, which is 8079:8079. It can be mapped to some other ports if needed.

1.5 Summary

...

After the configuration file, we can boot them all up with a simple "docker-compose up".

This demo sock-shop website was deployed on services provided by Aliyun®. It is currently available at http://101.132.128.124:8079.

The image can be referred at "./images/overall.png".

2 Part 2: Correctness Checking

2.1 Manually Checking

which is not very reliable

Firstly, I went to the website manually, imitating to be a normal user, registering a new account with all fake informations, etc. In a word, it works perfectly good as far as I see. But that's not even a way you would want to test your website. So let's move on to the next part.

2.2 Computer Testing

well let's see

According to the documentation, we should use “dplsming/sockshop-test:0.1” image to test our lame website.

And as you can see, it works perfectly well.

The image can be referred at “./images/correctness.png”.

3 Part 3: Load test

via the provided “load-test.py” scripts

The result .csv file will be presented at “./misc/out.csv”.

According to the results, I drew some images to explore the relations between them.

The graph image can be referred at “./images/graph.png”.

Seeing that graph, we can point out that the throughput increases as the load pressures higher. However when the load pressure reaches 50 or so, the throughput stops increasing, and keeps somehow steadily around 70.

The average responding time also shows the same rule as the throughput.

I suppose that phenomenon is caused by the limited calculating power. When the load is not adequate, the throughput will be small due to the limited visiting time. And the average responding is quite fast since there are no much requests to serve.

However as the load pressures higher, the throughput gradually got satisfied. However, it’s limited by the machine power and couldn’t go higher, then.

The average responding time slows down, too. But it won’t always goes higher, since it’s the average time for each request. Since the throughput goes on steady, it should be on steady, too.

Unlike those two variables, the 95% and 99% error rate goes higher and obviously without a limit. That’s because as the pressure goes on higher, the race condition error occurs more often, and more memory page fault were raised, too. Plus they will not be limited anyhow, and as the load goes on higher, the error rate will goes on higher as response.