Grenoble INP – ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Jhipster Project

Microservices Architectures - Practices with JHipter

Realized by:
Ismail BARKANI
Mehdi SAOUDI
Youssef TENNIA

Supervised by:
Pr. Didier DONSEZ

Contents

Repository informations

Repository Link:

- ▷ *microservice-online-store*

Contributors:

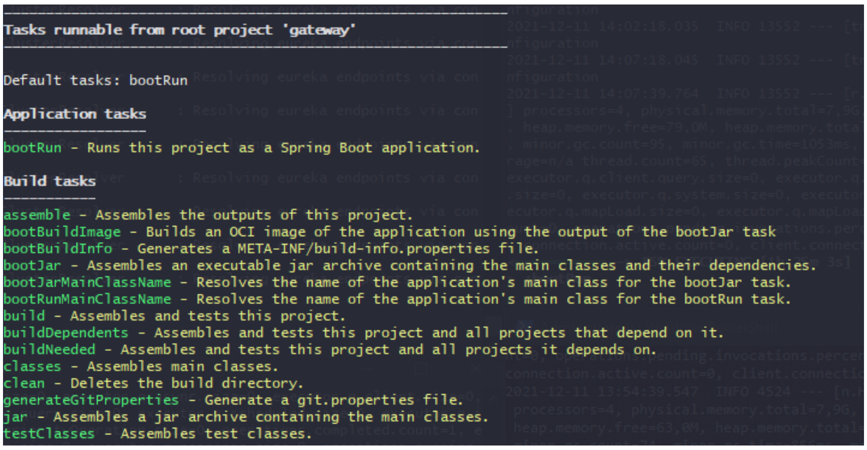
- ▷ Ismail Barkani (Github Id: *IsmailBarkani*, Email: ismail.barkani@grenoble-inp.org)
- ▷ Saoudi Mehdi (Github Id: *SaoudiMehdi*, Email: mehdi.saoudi@grenoble-inp.org)
- ▷ Youssef Tennia (Github Id: *TenniaYoussef*, Email: youssef.tennia@grenoble-inp.org)

CHAPTER 1 Deployment of online-store using Docker Compose

In this Section, we tried to deploy our microservices application using Docker Compose, we started by containerizing each microservice artifact into a docker image.

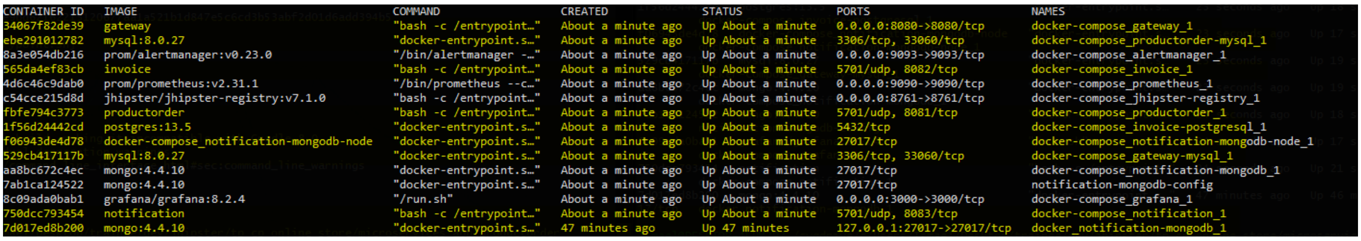
Notice :

Working with the last version of Jhipster **7.4.1**, we noticed that during the generation of a microservice using gradle, the only available building task is *bootJar* (screenshot below), also it generates a *JIB* configuration under *gradle/docker.gradle* which helps us to dockerize a microservice without using a Docker deamon : ***./gradlew bootJar -Pprod jibDockerBuild***



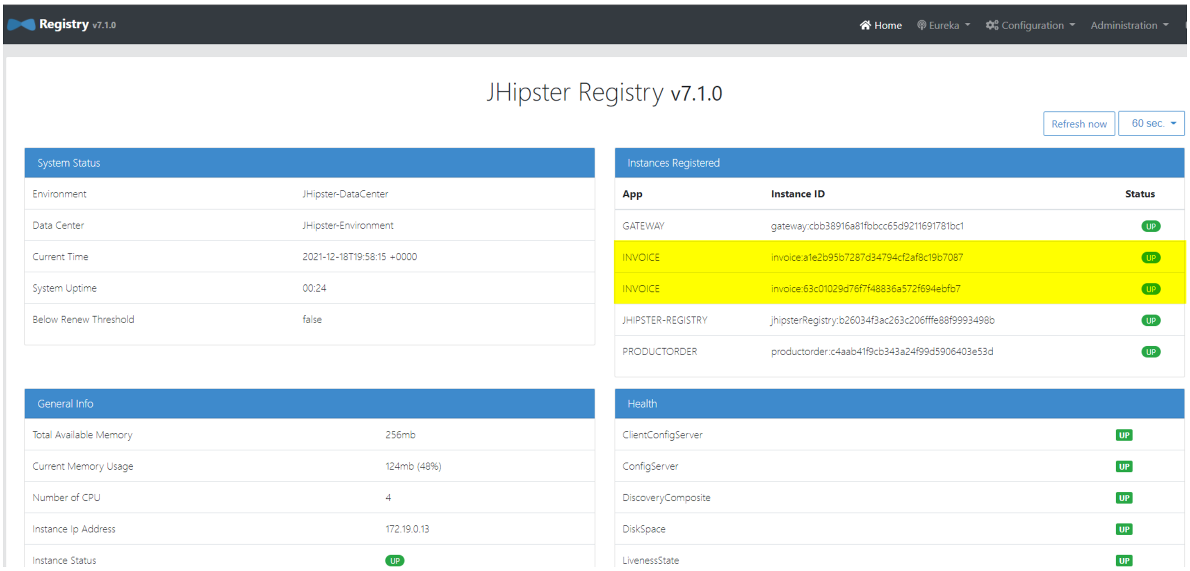
```
Tasks runnable from root project 'gateway'

Default tasks: bootRun

Application tasks
-----------------
bootRun - Runs this project as a Spring Boot application.

Build tasks
-----------
assemble - Assembles the outputs of this project.
bootBuildImage - Builds an OCI image of the application using the output of the bootJar task
bootBuildInfo - Generates a META-INF/build-info.properties file.
bootJar - Assembles an executable jar archive containing the main classes and their dependencies.
bootJarMainClassName - Resolves the name of the application's main class for the bootJar task.
bootRunMainClassName - Resolves the name of the application's main class for the bootRun task.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
generateGitProperties - Generate a git.properties file.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles test classes.
```

Afterwards, using Jhipster, we generated the Docker Compose manifest for all microservices and under the same root directory:

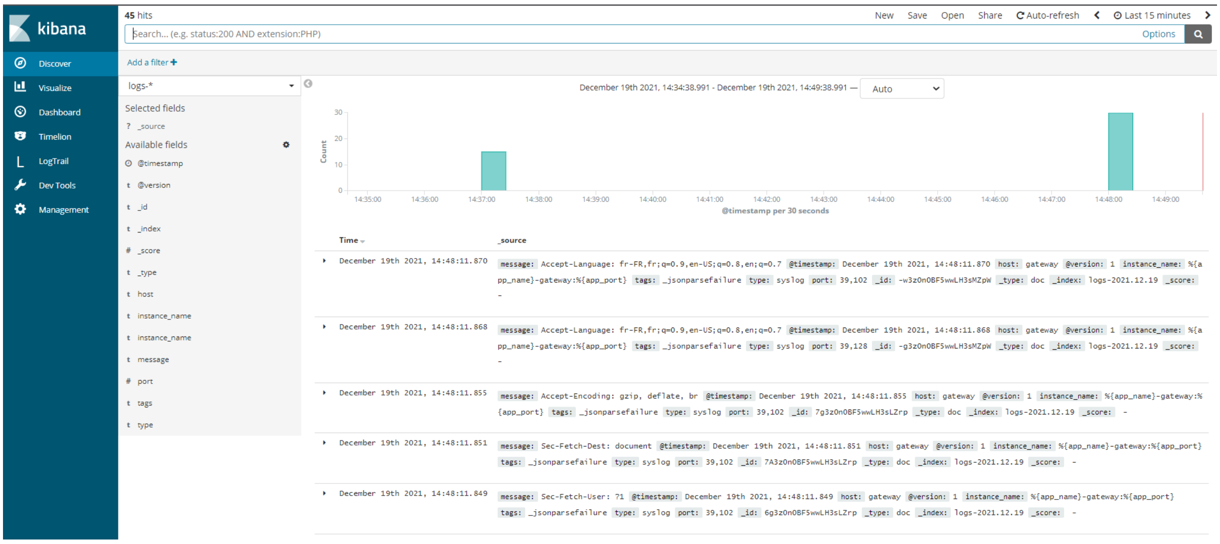


CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
34067f82de39	gateway	"bash -c /entrypoint…"	About a minute ago	Up About a minute	0.0.0.0:8080->8080/tcp	docker-compose_gateway_1
ebe291012782	mysql:8.0.27	"docker-entrypoint.s…"	About a minute ago	Up About a minute	3306/tcp, 33060/tcp	docker-compose_productorder-mysql_1
8a3e054db216	prom/alertmanager:v0.23.0	"/bin/alertmanager -…"	About a minute ago	Up About a minute	0.0.0.0:9093->9093/tcp	docker-compose_alertmanager_1
565da4ef83cb	invoice	"bash -c /entrypoint…"	About a minute ago	Up About a minute	5701/udp, 8082/tcp	docker-compose_invoice_1
4d6c46c9dab0	prom/prometheus:v2.31.1	"/bin/prometheus --c…"	About a minute ago	Up About a minute	0.0.0.0:9090->9090/tcp	docker-compose_prometheus_1
c54cce215d8d	jhipster/jhipster-registry:v7.1.0	"bash -c /entrypoint…"	About a minute ago	Up About a minute	0.0.0.0:8761->8761/tcp	docker-compose_jhipster-registry_1
fbfe794c3773	productorder	"bash -c /entrypoint…"	About a minute ago	Up About a minute	5701/udp, 8081/tcp	docker-compose_productorder_1
1f56d24442cd	postgres:13.5	"docker-entrypoint.s…"	About a minute ago	Up About a minute	5432/tcp	docker-compose_invoice-postgresql_1
f06943de4d78	docker-compose_notification-mongodb-node	"docker-entrypoint.s…"	About a minute ago	Up About a minute	27017/tcp	docker-compose_notification-mongodb-node_1
529cb417117b	mysql:8.0.27	"docker-entrypoint.s…"	About a minute ago	Up About a minute	3306/tcp, 33060/tcp	docker-compose_gateway-mysql_1
aa8bc672c4ec	mongo:4.4.10	"docker-entrypoint.s…"	About a minute ago	Up About a minute	27017/tcp	docker-compose_notification-mongodb_1
7ab1ca124522	mongo:4.4.10	"docker-entrypoint.s…"	About a minute ago	Up About a minute	27017/tcp	notification-mongodb-config
8c09ada0bab1	grafana/grafana:8.2.4	"/run.sh"	About a minute ago	Up About a minute	0.0.0.0:3000->3000/tcp	docker-compose_grafana_1
750dcc793454	notification	"bash -c /entrypoint…"	About a minute ago	Up About a minute	5701/udp, 8083/tcp	docker-compose_notification_1
7d017ed8b200	mongo:4.4.10	"docker-entrypoint.s…"	47 minutes ago	Up 47 minutes	127.0.0.1:27017->27017/tcp	docker_notification-mongodb_1

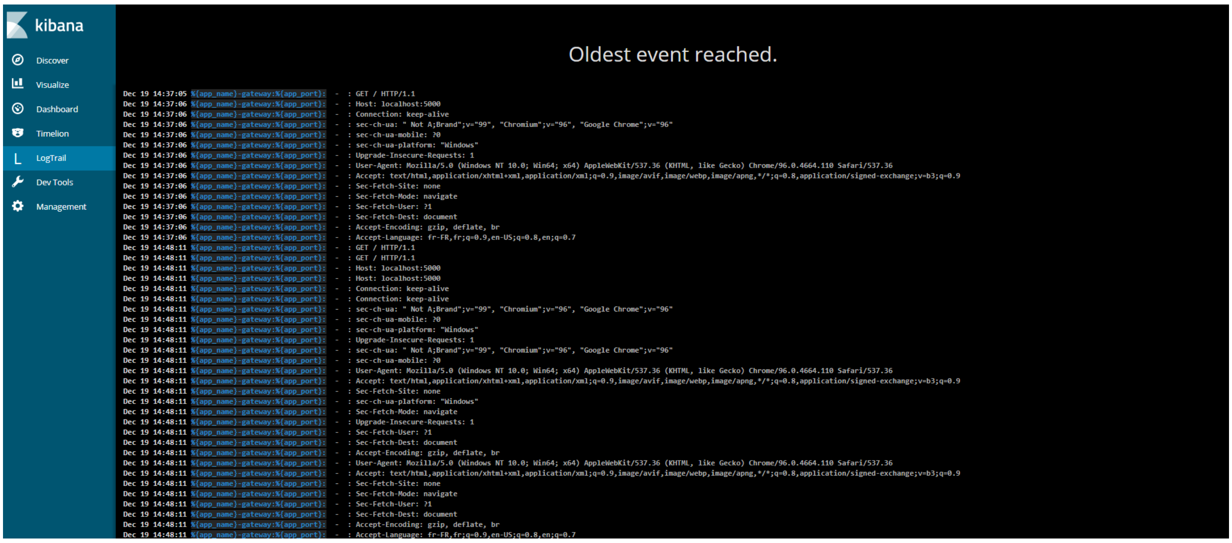
In order to meet changin demand, we scaled the invoice microservice up to 2 replicas, looking to jhipster registry on *localhost:8761* we can list each registred microservice with its replica:



In addition, for streaming logs and metrics we used the open-source application *Jhipster Console*. At first, we changed some settings on *application-dev.yml* (as we use the spring dev profile) by enabling *metrics.logs* and *logstach.logs* for all microservices including the gateway service. This modification will enable microservices to transfer logs and metrics to Jhipster Console running on port 5601. After re-deploying all microservices, we get the following metrics visualization:

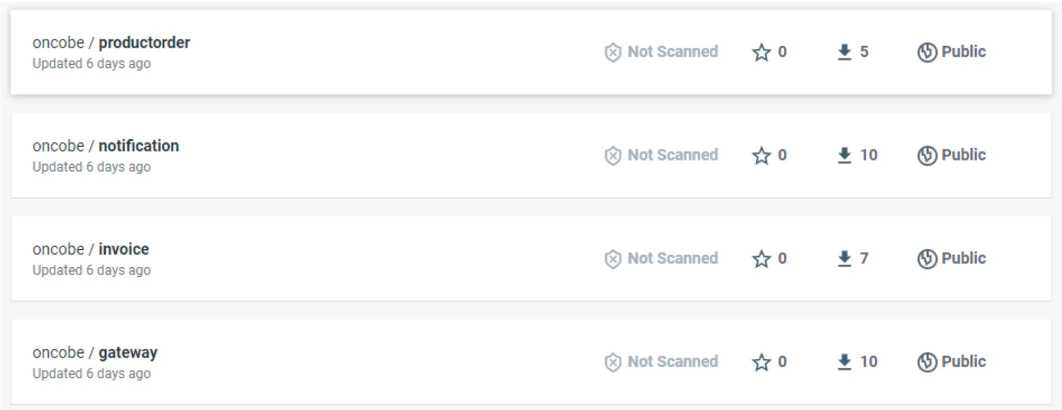


Also logs of all deployed services:



CHAPTER 2 Deployment of online-store with Kubernetes On GCP

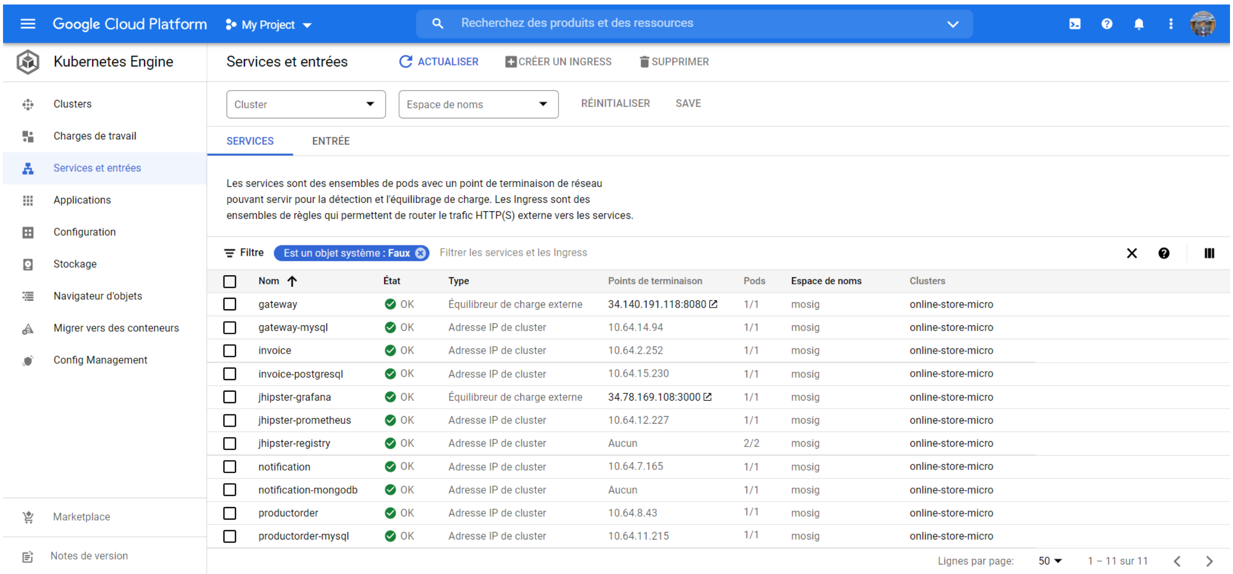
In this section, we deployed our microservices on GCP using Kubernetes as an orchestrator. The process of deploying with kubernetes is one of the hardest option, but Jhipster simplifies this process by generating the Kubernetes manifests for each microservices, which are basically files that describe how Kubernetes will deploy and manage our services.
These files point to public docker images, so we pushed each microservice image into our common docker registry named *oncobe*:



After generating all manifests in order to run our microservices-based application, we created a Kubernetes cluster in GCP using the installed SDK for gcp.
Also we used the medium machin type *–machine-type n1-standard-2*. After running this command:
gcloud container clusters create online-store-micro –machine-type n1-standard-2
We got the following cluster up and running:

```
NAME                LOCATION        MASTER_VERSION    MASTER_IP        MACHINE_TYPE    NODE_VERSION      NUM_NODES  STATUS
online-store-micro  europe-west1-b  1.21.5-gke.1302   35.241.207.215   n1-standard-2   1.21.5-gke.1302   3          RUNNING
```

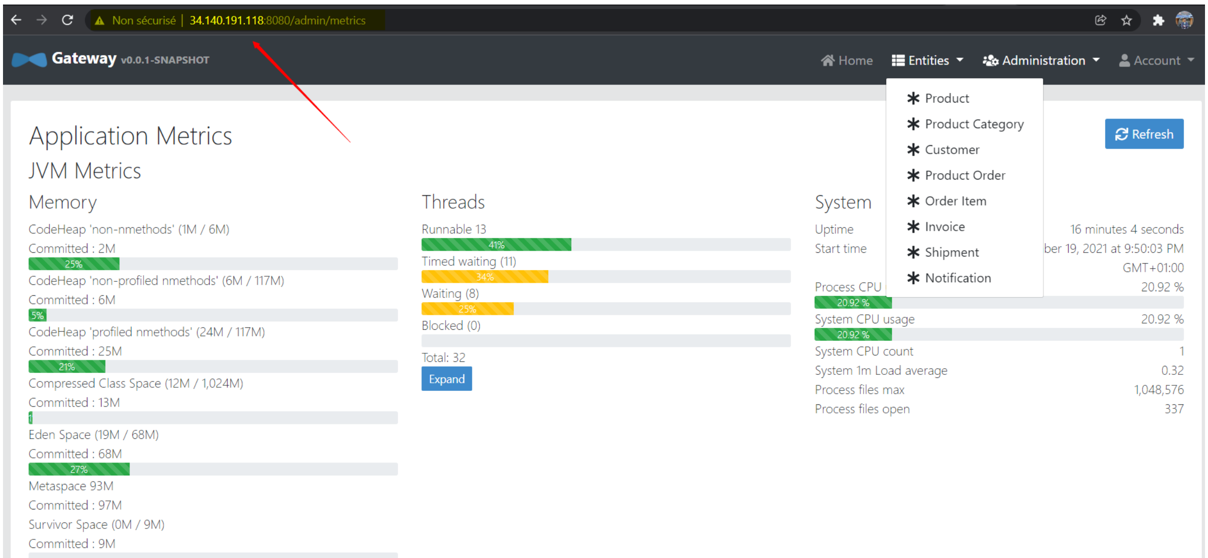
We can check now all running status of pods after deploying microservices on google cloud platform :



In addition, we can get the accessible external IP of our application store:

```
$ kubectl get svc gateway -n mosig
NAME      TYPE           CLUSTER-IP    EXTERNAL-IP      PORT(S)          AGE
gateway   LoadBalancer   10.64.4.239   34.140.191.118   8080:30088/TCP   15m
```

Also we can use this IP in order to access to the application through the gateway service on 8080, and display some metrics with a nice dashboard



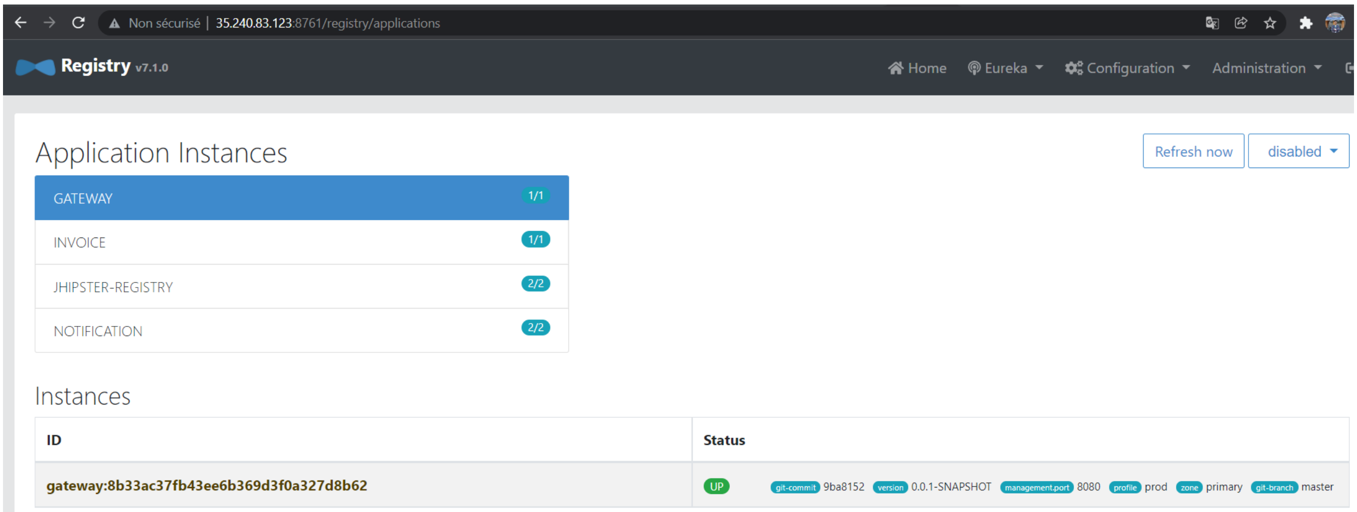
Moreover, one of the most interesting advantages with microservices-based application is the scalability of microservices (horizontal and vertical), now we add the scalability option to a specific microservices instead of doing it to the whole application (case of monolithic). For instance lets scale up the service of notification up to two replicas using the following commande :

```
$ kubectl scale deployment/notification --replicas=2 -n mosig
deployment.apps/notification scaled
```

As we see under the *Services & Ingress* we got two pods for notifications microservies:

Nom	État	Type	Points de terminaison	Pods	Espace de noms	Clusters
gateway	OK	Équilibreur de charge externe	34.140.191.118:8080	1/1	mosig	online-store-micro
gateway-mysql	OK	Adresse IP de cluster	10.64.14.94	1/1	mosig	online-store-micro
invoice	OK	Adresse IP de cluster	10.64.2.252	1/1	mosig	online-store-micro
invoice-postgresql	OK	Adresse IP de cluster	10.64.15.230	1/1	mosig	online-store-micro
jhipster-grafana	OK	Équilibreur de charge externe	34.78.169.108:3000	1/1	mosig	online-store-micro
jhipster-prometheus	OK	Adresse IP de cluster	10.64.12.227	1/1	mosig	online-store-micro
jhipster-registry	OK	Adresse IP de cluster	Aucun	2/2	mosig	online-store-micro
notification	OK	Adresse IP de cluster	10.64.7.165	1/2	mosig	online-store-micro
notification-mongodb	OK	Adresse IP de cluster	Aucun	1/1	mosig	online-store-micro

And by accessing to the service registry on 32.240.83.123:8761 we can see that we have two services of notification which are registered:

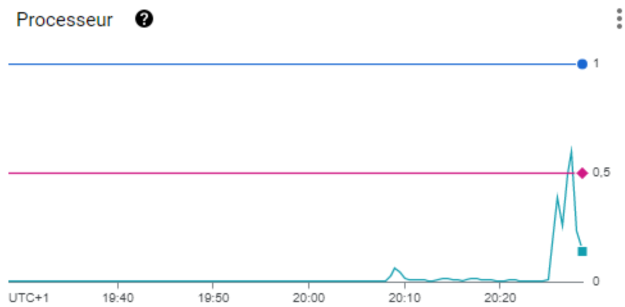


CHAPTER 3 Load Injection tests with Gatling

In this section we will test the scalability property of our microservice *notification, first lets enable the autoscaling policy using the following commande:*

```
C:\Users\Ismael\Desktop\trainee\dev\jhipster\tp_cp_online_store\microservices_app>kubectl autoscale deployment notification --cpu-percent=50 --min=1 --max=3 -n mosig
horizontalpodautoscaler.autoscaling/notification autoscaled
```

Here we have specified the threshold limit of CPU percentage in which the pod will be replicated and handle the high load of requests on notification microservices. We have specified also the maximum replicas allowed which is 3.
As you see in the following screenshot, when we launched *Gatling*, the notification microservices starts receiving a lot of requests, which increased the CPU consumption.
Notice that when the CPU consumption was around 50% it starts decreasing gradually, which is a result of scaling up the pod:

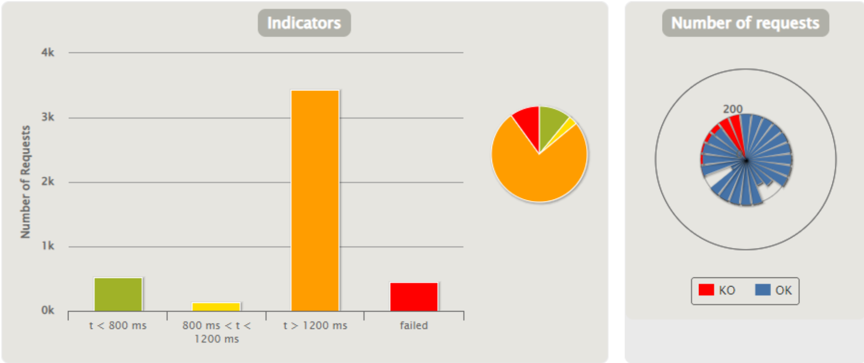


Here we got the maximum replica (max=3) after launching *Gatling*:

Pods gérés

Révision	Nom	État	Redémarrages	Créées le ↑
1	notification-b5bff9659-6m9hp	Running	1	25 déc. 2021, 15:58:48
1	notification-b5bff9659-vnd8f	Unschedulable	0	25 déc. 2021, 20:26:19
1	notification-b5bff9659-4wgkg	Unschedulable	0	25 déc. 2021, 20:26:19

Here is a summary of Gatling report (you can find the full report in our git repository), we can notice that after a while, responses time start to be relatively slow (> 1200 ms), and that because of the number of users (200 users at a time), so maybe we have to increase the maximum replica of our autoscaling policy.


Indicators
4k
3k
2k
1k
0k
Number of Requests
t < 800 ms
800 ms < t < 1200 ms
t > 1200 ms
failed
Number of requests
200
KO
OK