CLD Lab 05: Kubernetes

Auteurs: Edwin Haeffner, Arthur Junod Group: L5GrE Date: 10 May, 2024

Subtask Minikube :

> Document any difficulties you faced and how you overcame them. Copy the object descriptions into the lab report.

The only difficulty I got was when I was trying to post anything on the TodoV2 website, it wouldn't actually post anything.

By doing `kubectl logs -f frontend`

I was having this issue :

```
POST /api/todos 500 21.092 ms - -
[HPM] PROXY ERROR: ENOTFOUND. localhost -> http://api-
service:8081/api/todos
```

The value on my API_ENDPOINT_URL was wrong. I did not use the same name that I gave to my `api-svc.yaml` but changing it to the right name fixed the issue !

Object descriptions :

We're only giving you the one we had to create :

`api-svc.yaml`

```
apiVersion: v1
kind: Service
metadata:
  labels:
    component: api
  name: api-svc
spec:
  ports:
    - port: 8081
      targetPort: 8081
      name: api
  selector:
    app: todo
    component: api
  type: ClusterIP
```

`frontend-pod.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    component: frontend
    app: todo
spec:
  containers:
    - name: frontend
      image: icclabcna/ccp2-k8s-todo-frontend
      ports:
        - containerPort: 8080
      env:
        - name: API_ENDPOINT_URL
          value: http://api-svc:8081
```

SUBTASK 2.4 - DEPLOY THE TODO-FRONTEND SERVICE

> Document any difficulties you faced and how you overcame them. Copy the object descriptions into the lab report (if they are unchanged from the previous task just say so).

The process was pretty straight forward, so no difficulties here.

Object descriptions :

No changes done since last task.

> Take a screenshot of the cluster details from the GKE console. Copy the output of the kubectl describe command to describe your load balancer once completely initialized.

After doing this command : `kubectl describe service frontend`

We get this :

```
Name:                     frontend
Namespace:                default
Labels:                   app=todo
                          component=frontend
Annotations:              cloud.google.com/neg: {"ingress":true}
Selector:                 app=todo,component=frontend
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.106.60.57
IPs:                      10.106.60.57
LoadBalancer Ingress:     34.88.154.77
Port:                     <unset>  80/TCP
```

```
TargetPort:               8080/TCP
NodePort:                 <unset>  31600/TCP
Endpoints:                10.60.0.5:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age    From                Message
  ----    ------                ----   ----                -------
  Normal  EnsuringLoadBalancer  2m33s  service-controller  Ensuring load
balancer
  Normal  EnsuredLoadBalancer   110s   service-controller  Ensured load
balancer
```

We can connect to the application by inserting the IP of the load balancer in the url.

Screenshot of the cluster details

Cluster basics

Name	gke-cluster-1	
Location type	Zonal	
Control plane zone	europe-north1-a	
Default node zones	europe-north1-a	
Release channel	Regular channel	UPGRADE AVAILABLE
Version	1.28.7-gke.1026000	
Total size	2	
External endpoint	35.228.113.196 Show cluster certificate	
Internal endpoint	10.166.0.2 Show cluster certificate	

TASK 3 - ADD AND EXERCISE RESILIENCE

> Use only 1 instance for the Redis-Server. Why?

Because if we have multiple instances, the database won't be linked together, which would display different data on the frontend if they are not connected to the same instance. We would need to have a manager that can replicate the data across all instances or use a system of replicas that can be read from but not written to, with one master instance.

> What happens if you delete a Frontend or API Pod? How long does it take for the system to react?

API Pod -> It took less than two seconds to create a new one. But the pod itself can take a while to terminate. It instantly create a new pod without waiting for the other one to be terminated.

While the pod is being deleted and a new one is being created, we can still post TODOs and the page reloads fine.

Frontend Pod -> It seems to be working the same way as deleting the API Pod, terminating a frontend pod is taking way less time though.

When reloading the page, it was unavailable once but it got back to working fairly quickly (~2 seconds).

> What happens when you delete the Redis Pod?

It seems to do the same as the other pods when we look at the terminal but when reloading the page, we lose the previously stored TODOs... It makes sense since we're storing our data in ephemeral storage on the redis pod. The issue now is that we cannot post anything anymore. The only way to restore the website's function was to restart everything. From looking into the browser's console, the issue seemed to be linked with IDs. But even without posting anything (and therefore never updating the IDs), terminating only the redis pod would lead to the same issue.

> How can you change the number of instances temporarily to 3? Hint: look for scaling in the deployment documentation

```
kubectl scale deployment api --replicas=3
```

Instead of api we can also put frontend or redis. (shouldn't put redis tho !!!)

> What autoscaling features are available? Which metrics are used?

1. Horizontally using a HorizontalPodAutoscaler (HPA) and the autoscaling is based on various metrics like CPU utilization, memory usage, and custom metrics.
2. Vertically using a VerticalPodAutoscaler (VPA). Unlike the HPA, the VPA doesn't come with Kubernetes by default, but is a separate project that can be found on GitHub.
3. There's also autoscalers based on cluster sizes : Cluster Proportional Autoscaler and Cluster Proportional Vertical Autoscaler
4. It is also possible to scale workloads based on events, for example using the Kubernetes Event Driven Autoscaler (KEDA).
5. And finally we can also do this based on a schedule, for example in order to reduce resource consumption during off-peak hours.

> How can you update a component? (see “Updating a Deployment” in the deployment documentation)

You can use this command to update the image of a deployment:

```
kubectl set image deployment/<deployment name> <container name>=<new image>
```

or alternatively directly edit the deployment:

```
kubectl edit deployment/<deployment name>
```

> Put autoscaling in place and load-test it

Apply HPA : `kubectl autoscale deployment frontend --cpu-percent=30 --min=1 --max=4`

Vegeta attack :

~25 seconds after the start of the attack, we can see that the cpu is detecting that it's being utilized by doing this command : `kubectl get hpa` ->

```
NAME       REFERENCE              TARGETS    MINPODS   MAXPODS   REPLICAS
AGE
frontend   Deployment/frontend   405%/30%   1         4         2
3m18s
```

And in the `deploy --watch`, we see that the amount of replicas increased :

```
api        1/2     2            1           3s
frontend   1/2     2            1           15s
redis      1/1     1            1           21s
api        2/2     2            2           23s
frontend   2/2     2            2           40s
frontend   2/4     2            2           4m53s
frontend   2/4     2            2           4m53s
frontend   2/4     2            2           4m53s
frontend   2/4     4            2           4m53s
frontend   3/4     4            3           4m56s
frontend   4/4     4            4           4m58s
```

Result of the vegeta test :

```
Requests      [total, rate, throughput]         3000, 50.02, 49.27
Duration      [total, attack, wait]             1m1s, 59.98s, 907.702ms
Latencies     [min, mean, 50, 90, 95, 99, max]  82.471ms, 133.696ms,
127.163ms, 139.221ms, 170.589ms, 453.494ms, 1.008s
Bytes In      [total, mean]                     1893000, 631.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           100.00%
Status Codes  [code:count]                      200:3000
Error Set:
```

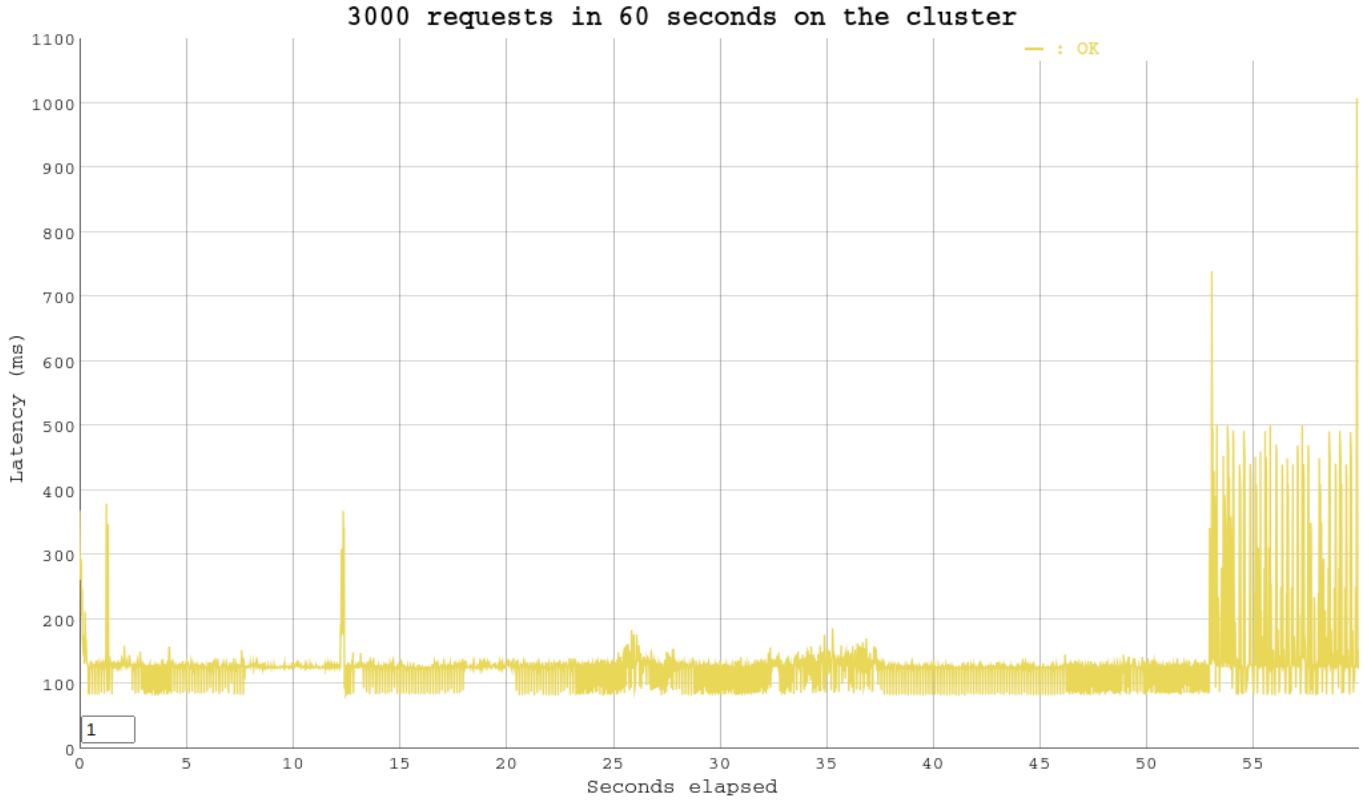
But then, at the end of the test, all of the pods shut down :

```
api        1/2     2            1           6m12s
frontend   3/4     4            3           6m24s
frontend   2/4     4            2           6m24s
redis      0/1     1            0           7m26s
frontend   1/4     4            1           7m20s
api        0/2     2            0           7m8s
```

```
frontend   0/4     4            0           7m20s
redis      1/1     1            1           10m
frontend   1/4     4            1           10m
api        1/2     2            1           10m
frontend   2/4     4            2           10m
```

It took almost 3 minutes to recover and while everything was down, the website was unreachable.

Checking in the Kubernetes node detail, it would show us that the container's runtime was down. Maybe 3000 requests in 60 seconds did put too much strain on the node. 😛



Now looking at the attack's response graph, we see that nearing the end, the requests are struggling more and more, the latest request taking an entire second to be answered. But thanks to the auto scalling, it was able to answer to every requests.

Object descriptions :

frontend-deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    component: frontend
    app: todo
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      component: frontend
      app: todo
  template:
    metadata:
      labels:
        component: frontend
        app: todo
    spec:
      containers:
        - name: frontend
          image: icclabcna/ccp2-k8s-todo-frontend
          ports:
            - containerPort: 8080
          env:
            - name: API_ENDPOINT_URL
              value: http://api-svc:8081
          resources:
            requests:
              cpu: 10m
```

api-deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  labels:
    component: api
    app: todo
spec:
  replicas: 2
  selector:
    matchLabels:
      component: api
      app: todo
  template:
    metadata:
      labels:
        component: api
        app: todo
    spec:
      containers:
        - name: api
          image: icclabcna/ccp2-k8s-todo-api
          ports:
            - containerPort: 8081
          env:
            - name: REDIS_ENDPOINT
              value: redis-svc
            - name: REDIS_PWD
```

```
              value: ccp2
```

redis-deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
  labels:
    component: redis
    app: todo
spec:
  replicas: 1
  selector:
    matchLabels:
      component: redis
      app: todo
  template:
    metadata:
      labels:
        component: redis
        app: todo
    spec:
      containers:
        - name: redis
          image: redis
          ports:
            - containerPort: 6379
          args:
            - redis-server
            - --requirepass ccp2
            - --appendonly yes
```