

Projet Conteneurisation orchestration



Conteneurisation et orchestration
Projet Docker

Sommaire :

- 1. Analyse des fichiers**
- 2. Déploiement**
- 3. Bilan de santé**
- 4. Sécurité**
- 5. Logs**
 - a) Elasticsearch**
 - b) Kibana**
 - c) Fluentd**
- 6. Déploiement automatisé**

1. Analyse des fichiers

Applications

L'objectif est le déploiement, la maintenance et la mise à jour de différents micro-services REST et applications web dans un environnement conteneurisé, avec un orchestrateur Kubernetes afin d'assurer la haute disponibilité des apps.

Ce qui est fournis

Pour ce projet, je jouerais le rôle du développeur. Vous pourrez donc me poser des questions sur le fonctionnement des apps, ce qu'elles ont besoin en terme de ressources et de liaisons. Si il le faut, je pourrais modifier le code des applications afin de les adapter à votre système (tant que ça reste de la configuration). Les applications sont disponibles sur le [git](<https://github.com/bart120/m1cloud/tree/master/projet/appscore>) du cours.

Description des apps:

- web => application web qui consomme les services applicants.api et jobs.api
- applicants.api => service API REST fournissant des données d'une base de données.
- identity.api => service API REST fournissant une authentification
- jobs.api => service API REST fournissant des données d'une base de données.
- sql.data => base données SQL Server et données
- rabbitmq => fournis une communication entre les services API REST

Afin de répondre au mieux au sujet ci-contre nous avons commencé par analyser les fichiers spécifiés. Le dossier dont il s'agit ici est composé de 153 fichiers et de 85 sous-dossiers. Il a pour objectif la mise en place d'une application composée de plusieurs services.

Et pour cela il est composé de 6 grandes sections.

Les 3 services d'applications (jobs, applicants, Identity).

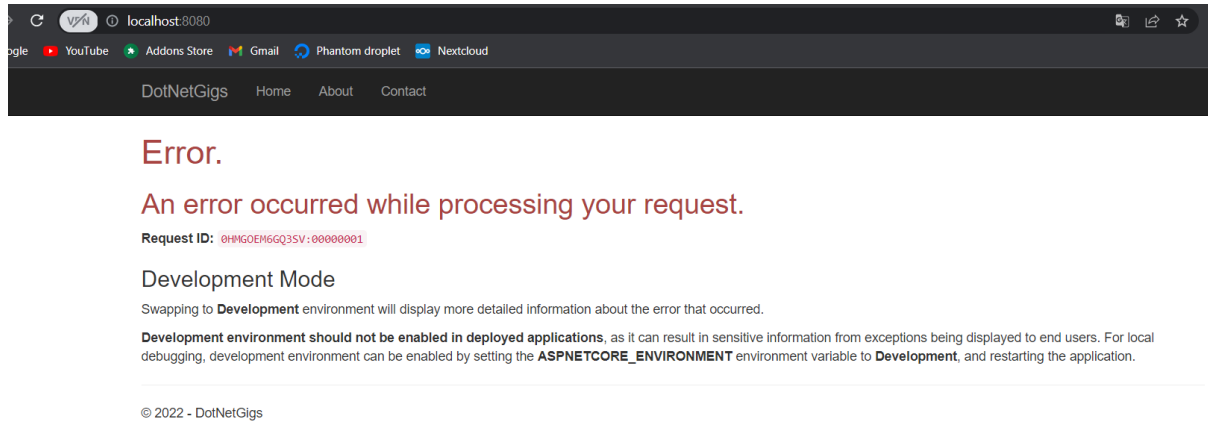
Un serveur web, un serveur rabbit MQ.

Et enfin une base de données SQL.

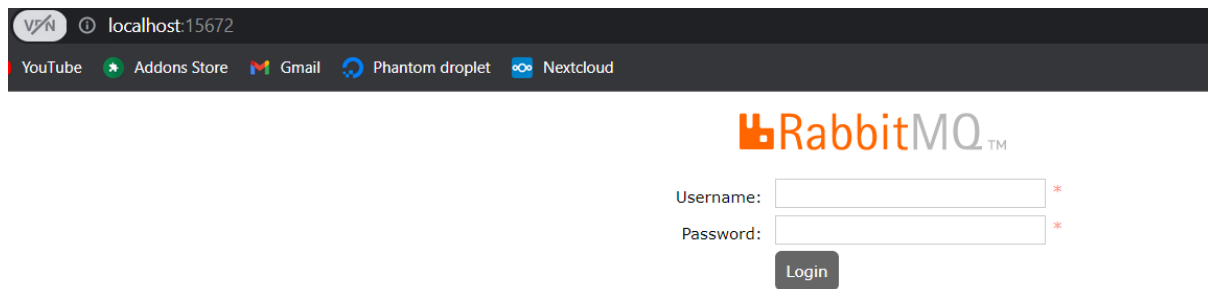
Ils devront tous être fonctionnels, et servir au bon fonctionnement des applications.

- Première étape : On lance un docker-compose up du fichier docker-compose.yml
- Deuxième étape : On vérifie que ça fonctionne en se connectant aux conteneur web par exemple. On obtient ce résultat.

Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA



- Troisième étape : On vérifie aussi que le rabbit mq est fonctionnel. C'est le cas, on obtient le résultat suivant



Et comme on peut le voir sur le fichier docker-compose.yml chaque conteneur a une adresse locale prédéfinie.

On comprend alors que l'on va devoir déployer tout ça avec kubernetes. C'est la partie déploiement du projet.

2. Déploiement

Pour effectuer le déploiement on commence donc par créer tous les fichiers nécessaires. On comprend qu'il va falloir produire un fichier yaml pour le déploiement de ; applicants ; Identity ; jobs ; rabbit ; user data ; web ; Sans oublier l'ingress.

Ci-dessous le détail de tous les fichiers :

Applicants

```
#deployment applicants

apiVersion: apps/v1
kind: Deployment
metadata:
  name: applicants-deployment
  labels:
    app: applicants
spec:
  replicas: 1
  selector:
    matchLabels:
      app: applicants-pod
  template:
    metadata:
      labels:
        app: applicants-pod
    spec:
      containers:
        - name: applicants-cont
          image: tmezgouria/applicants:applicants.api-latest
          ports:
            - containerPort: 80
          resources:
            requests:
              memory: 500Mi
              cpu: 3m
            limits:
              memory: 1500Mi
              cpu: 6m
          restartPolicy: Always
---
#service applicants

apiVersion: v1
kind: Service
metadata:
  name: applicants-service
```

Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA

```
spec:
  selector:
    app: applicants-pod
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP
```

Identity

```
#deployment identity

apiVersion: apps/v1
kind: Deployment
metadata:
  name: identity-deployment
  labels:
    app: identity
spec:
  replicas: 1
  selector:
    matchLabels:
      app: identity-pod
  template:
    metadata:
      labels:
        app: identity-pod
    spec:
      containers:
        - name: identity-cont
          image: tmezgouria/identity:identity.api-latest
          ports:
            - containerPort: 80
          resources:
            requests:
              memory: 500Mi
              cpu: 3m
            limits:
              memory: 1500Mi
              cpu: 6m
          restartPolicy: Always
---

#service identity
```

Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA

```
apiVersion: v1
kind: Service
metadata:
  name: identity-service
spec:
  selector:
    app: identity-pod
  ports :
    - port: 80
      targetPort: 80
  type: ClusterIP
```

Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-core
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - http:
        paths:
          - path: /web
            pathType: Prefix
            backend:
              service:
                name: service-web
                port:
                  number: 80
#     - path: /
#       pathType: Prefix
#       backend:
#         service:
#           name: service-rabittmq
#           port:
#             number: 15672
```

Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA

Jobs

```
#deployment jobs

apiVersion: apps/v1
kind: Deployment
metadata:
  name: jobs-deployment
  labels:
    app: jobs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jobs-pod
  template:
    metadata:
      labels:
        app: jobs-pod
    spec:
      containers:
        - name: jobs-cont
          image: tmezgouria/jobs:jobs.api-latest
          ports:
            - containerPort: 80
          resources:
            requests:
              memory: 500Mi
              cpu: 3m
            limits:
              memory: 1500Mi
              cpu: 6m
          restartPolicy: Always
---

#service jobs

apiVersion: v1
kind: Service
metadata:
  name: jobs-service
spec:
  selector:
    app: jobs-pod
```


Abdoulaye BAGAYOKO

Taha MEZGOURIA

Winsley SAHA

```
ports:

- port: 80
  targetPort: 80
type: ClusterIP
```

Rabbit

```
#deployment Rabbitmq

apiVersion: apps/v1
kind: Deployment
metadata:
  name: rabbitmq-deployment
  labels:
    app: rabbitmq
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      containers:
        - name: rabbitmq-cont
          image: rabbitmq:3-management
          ports:
            - containerPort: 15672
      # resources:
      #   requests:
      #     memory: 500Mi
      #     cpu: 4m
      # limits:
      #   memory: 1500Mi
      #   cpu: 8m
      restartPolicy: Always

---

#service rabbitmq

apiVersion: v1
```

Abdoulaye BAGAYOKO

Taha MEZGOURIA

Winsley SAHA

```
kind: Service
```

```
metadata:
  name: rabbitmq-service
spec:
  selector:
    app: rabbitmq
  ports:
    - port: 80
      targetPort: 15672
  type: ClusterIP
```

User data

```
#deployment userdata

apiVersion: apps/v1
kind: Deployment
metadata:
  name: userdata-deployment
  labels:
    app: userdata
spec:
  replicas: 1
  selector:
    matchLabels:
      app: userdata-pod
  template:
    metadata:
      labels:
        app: userdata-pod
    spec:
      containers:
        - name: userdata-cont
          image: redis
          ports:
            - containerPort: 80
          resources:
            requests:
              memory: 500Mi
              cpu: 4m
            limits:
              memory: 1500Mi
              cpu: 8m
          restartPolicy: Always
```

Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA

```
---

#service userdata

apiVersion: v1
kind: Service
metadata:
  name: userdata-service
spec:
  selector:
    app: userdata-pod
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP
```

Web

```
#deployment web

apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
  labels:
    app: web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web-cont
          image: tmezgouria/web:web-latest
          ports:
            - containerPort: 80

---

#service web
```

Abdoulaye BAGAYOKO

Taha MEZGOURIA

Winsley SAHA

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: web-service
```

```
spec:
```

```
  selector:
```

```
    app: web
```

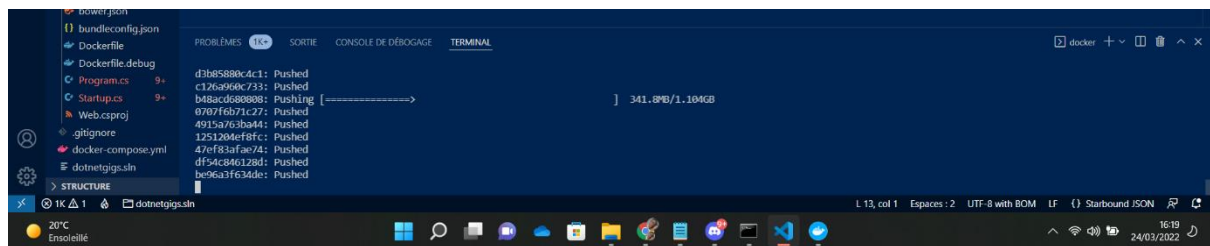
```
  ports:
```

```
    - port: 80
```

```
      targetPort: 80
```

```
  type: ClusterIP
```

Après les avoir créés on les a tous push sur le repo suivant : <https://hub.docker.com/u/tmezgouria>



Grâce à ces fichiers on a réussi à obtenir un résultat tangible mais présentant néanmoins quelques défauts. En effet on arrivait à accéder à la page web mais seulement en entrant l'adresse « localhost/ » les redirections étaient encore imparfaites.

Puis on s'est fait assister par des fichiers yaml déjà faits, octroyés par le professeur.

Un fichier « all.yml » et un fichier « ingress.yml ».

Ainsi naturellement, en faisant le déploiement, on a obtenu un résultat plutôt satisfaisant.

L'application est désormais fonctionnelle.

< Imprim écran page web fonctionnelle >

3. Bilan de santé

Afin d'établir le bilan de santé de notre cluster on commence par créer un namespace « monitoring ». Ensuite on crée un cluster-role qui va permettre de faire le lien entre le monitoring et les métriques kubernetes.

Cluster role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
    - nodes
    - nodes/proxy
    - services
    - endpoints
    - pods
  verbs: ["get", "list", "watch"]
- apiGroups:
  - extensions
  resources:
  - ingresses
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: default
  namespace: monitoring
```

Ensuite on crée la config-map.yml (voir repo GIT)

Déployer prometheus sur notre cluster. Pour le faire on a exécuté la cmd suivante : « kubectl apply -f prometheus-deployment/ » On suit les étapes suivantes pour cela :

Abdoulaye BAGAYOKO

Taha MEZGOURIA

Winsley SAHA

Prometheus deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus-deployment
  namespace: monitoring
  labels:
    app: prometheus-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus-server
  template:
    metadata:
      labels:
        app: prometheus-server
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus
          args:
            - "--storage.tsdb.retention.time=12h"
            - "--config.file=/etc/prometheus/prometheus.yml"
            - "--storage.tsdb.path=/prometheus/"
          ports:
            - containerPort: 9090
          resources:
            requests:
              cpu: 500m
              memory: 500M
            limits:
              cpu: 1

              memory: 1Gi
      volumeMounts:
        - name: prometheus-config-volume
          mountPath: /etc/prometheus/
        - name: prometheus-storage-volume
          mountPath: /prometheus/
      volumes:
        - name: prometheus-config-volume
          configMap:
            defaultMode: 420
            name: prometheus-server-conf
```

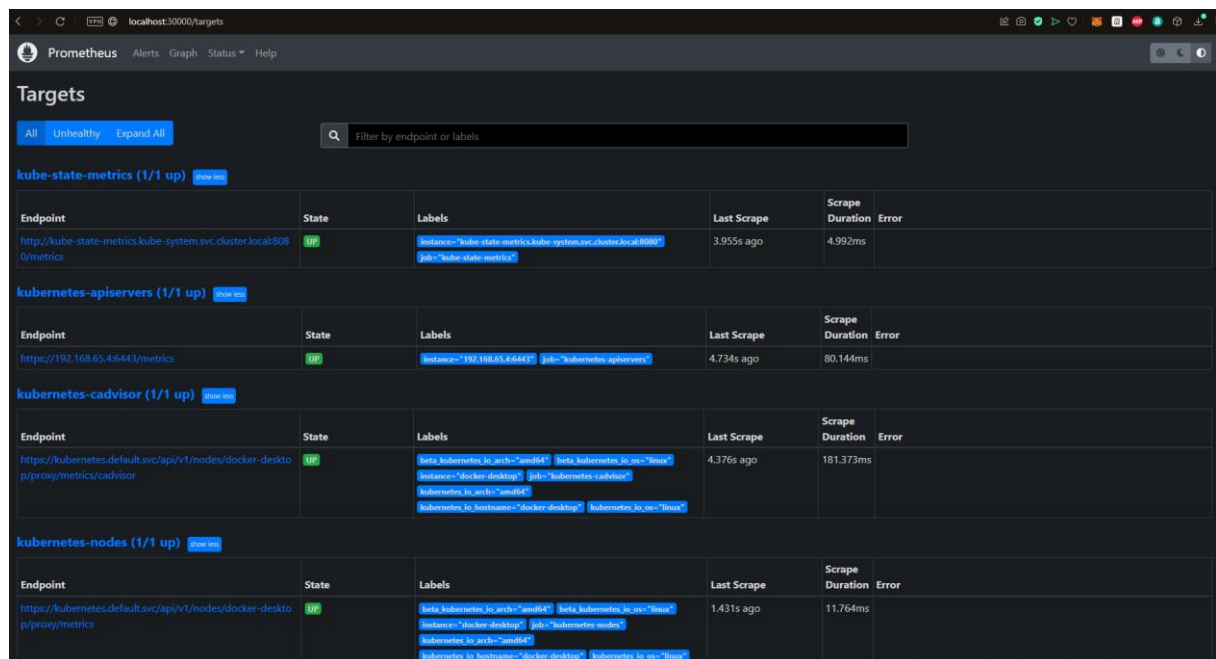
Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA

```
- name: prometheus-storage-volume
  emptyDir: {}
```

Prometheus service

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus-service
  namespace: monitoring
  annotations:
    prometheus.io/scrape: 'true'
    prometheus.io/port: '9090'
spec:
  selector:
    app: prometheus-server
  type: NodePort
  ports:
    - port: 8080
      targetPort: 9090
      nodePort: 30000
```

Suite à quoi le résultat final :



The screenshot shows the Prometheus web interface at localhost:3000/targets. The 'Targets' page displays a list of scraped targets, all of which are in the 'UP' state. The targets are grouped into four categories: kube-state-metrics, kubernetes-apiservers, kubernetes-cadvisor, and kubernetes-nodes. Each group shows the endpoint, state, labels, last scrape time, duration, and any errors.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
kube-state-metrics (1/1 up)					
http://kube-state-metrics.kube-system.svc.cluster.local:8080/metrics	UP	instance="kube-state-metrics.kube-system.svc.cluster.local:8080" job="kube-state-metrics"	3.955s ago	4.992ms	
kubernetes-apiservers (1/1 up)					
https://192.168.65.4:6443/metrics	UP	instance="192.168.65.4:6443" job="kubernetes-apiservers"	4.734s ago	80.144ms	
kubernetes-cadvisor (1/1 up)					
https://kubernetes.default.svc/api/v1/nodes/docker-desktop/prow/metrics/cadvisor	UP	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="docker-desktop" job="kubernetes-cadvisor" kubernetes_io_arch="amd64" kubernetes_io_hostname="docker-desktop" kubernetes_io_os="linux"	4.376s ago	181.373ms	
kubernetes-nodes (1/1 up)					
https://kubernetes.default.svc/api/v1/nodes/docker-desktop/prow/metrics	UP	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="docker-desktop" job="kubernetes-nodes" kubernetes_io_arch="amd64" kubernetes_io_hostname="docker-desktop" kubernetes_io_os="linux"	1.431s ago	11.764ms	

Comme on peut le voir sur l'image ci-dessus, La remontée des informations liées à Kubernetes métriques est un succès.

4. Sécurité

Sécurité

Vos apps devront être accessible uniquement par le protocole https de l'extérieur. Il vous faudra donc générer un [certificat](https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster) (auto-signé) SSL, et l'associer à votre cluster.

Nous commençons donc par créer les clés en question sur une machine ubuntu tierce.

À l'aide de la commande : `mkdir openssl && cd openssl.`

```
openssl req -x509 \  
    -sha256 -days 356 \  
    -nodes \  
    -newkey rsa:2048 \  
    -subj "/localhost/C=US/L=San Fransisco" \  
    -keyout rootCA.key -out rootCA.crt
```

Suite à cela on édite la clé privée de la sorte : `openssl genrsa -out server.key 2048`

(Voir fichier server.key sur le git)

Et on génère le fichier suivant `openssl req -new -key server.key -out server.csr -config csr.conf`

On édite également le fichier engendré.

Enfin on execute les commandes suivantes :

```
openssl x509 -req \  
    -in server.csr \  
    -CA rootCA.crt -CAkey rootCA.key \  
    -CAcreateserial -out server.crt \  
    -days 365 \  
    -sha256 -extfile cert.conf
```

On installe pour finir le certificat sur notre navigateur et c'est bon.

5. Logs

Logs

Les apps font remonter des logs dans la sortie standard. Afin de pouvoir les exploiter correctement il vous faudra mettre en place une pile complète (EFK ou ELK) afin de fournir une gestion de logs avancée.

Pour avoir un visuel sur la remontée des logs en procède à l'installation d'un environnement EFK.

a) Elasticsearch

On commence par installer Elasticsearch en suivant le procédé suivant.

On crée tout d'abord le fichier **es-svc.yml** on le déploie ensuite

```
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  labels:
    app: elasticsearch
spec:
  selector:
    app: elasticsearch
  clusterIP: None
  ports:
    - port: 9200
      name: rest
    - port: 9300
      name: inter-node
```

On crée également le fichier **es-sts.yml** qu'on créera avec la commande suivante : `kubectl create -f es-sts.yml`

On vérifie le déploiement en effectuant la cmd suivante : `kubectl port-forward es-cluster-0 9200:9200`

Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA

On est sûr que ça marche car on obtient le résultat suivant

```
localhost:9200
{
  "name": "es-cluster-0",
  "cluster_name": "k8s-logs",
  "cluster_uuid": "Tp59VdhpRcS25r8LKl0eRg",
  "version": {
    "number": "7.5.0",
    "build_flavor": "default",
    "build_type": "docker",
    "build_hash": "e9ccea4d68e2fac2275a3761849cbee64b39519f",
    "build_date": "2019-11-26T01:06:52.518245Z",
    "build_snapshot": false,
    "lucene_version": "8.3.0",
    "minimum_wire_compatibility_version": "6.8.0",
    "minimum_index_compatibility_version": "6.0.0-beta1"
  },
  "tagline": "You Know, for Search"
}
```

b) Kibana

L'implémentation de Kibana se fait plutôt simplement, On crée un fichier contenant le deployment et le service qu'on déploie ensuite.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kibana
  labels:
    app: kibana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kibana
  template:
    metadata:
      labels:
        app: kibana
    spec:
      containers:
        - name: kibana
          image: docker.elastic.co/kibana/kibana:7.5.0
      resources:
        limits:
```

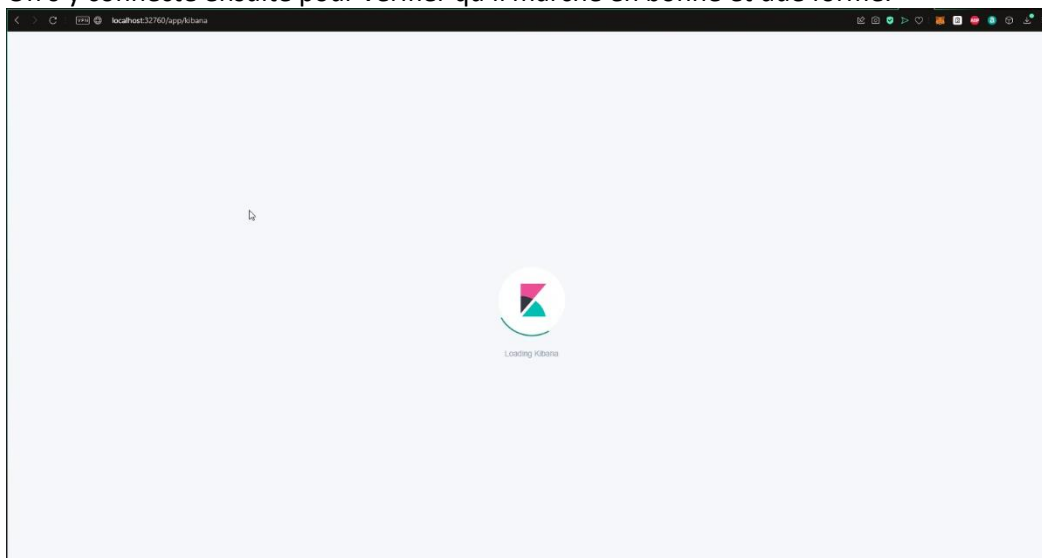
Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA

```
cpu: 1000m
  requests:
    cpu: 100m
  env:
    - name: ELASTICSEARCH_URL
      value: http://elasticsearch:9200
  ports:
    - containerPort: 5601

---

apiVersion: v1
kind: Service
metadata:
  name: kibana-np
spec:
  selector:
    app: kibana
  type: NodePort
  ports:
    - port: 8080
      targetPort: 5601
      nodePort: 32760
```

On s'y connecte ensuite pour vérifier qu'il marche en bonne et due forme.



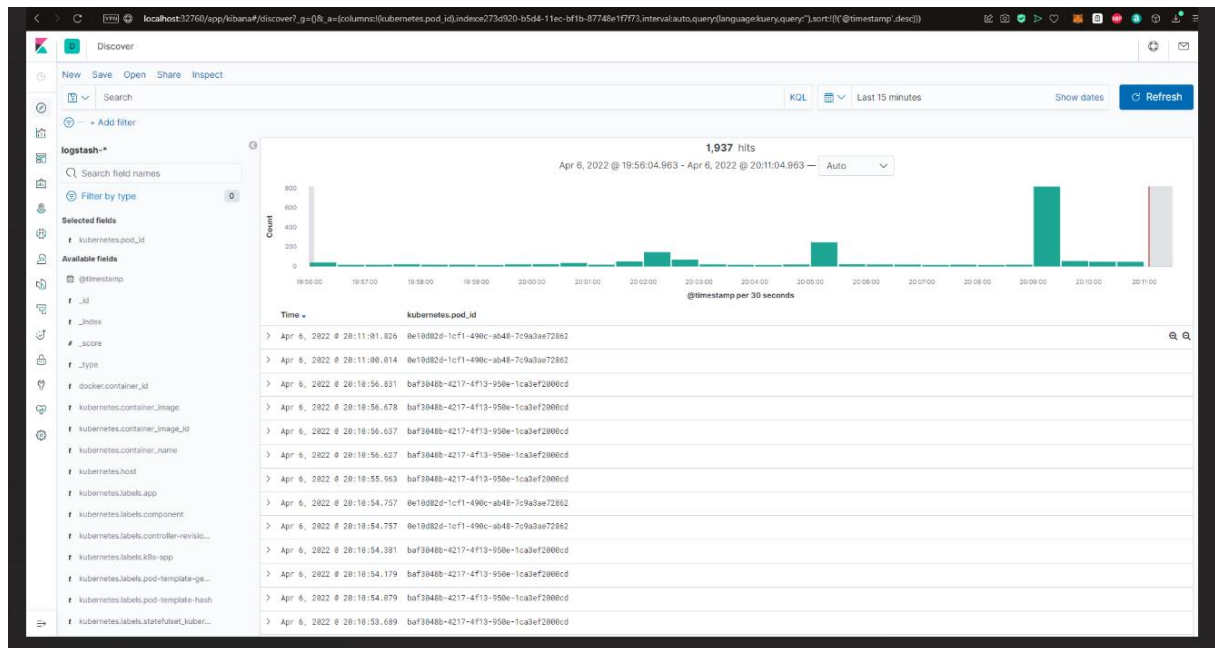
Abdoulaye BAGAYOKO
Taha MEZGOURIA
Winsley SAHA

c) Fluentd

Pour finir on installe Fluentd afin de permettre de faire le lien entre Kibana et ce qui se passe en temps réel et de remonter les logs.

On s'appuie sur l'image suivante : `image: fluent/fluentd-kubernetes-daemonset:v1.4.2-debian-elasticsearch-1.1`

Et on déploie un fichier complet.



On obtient une remontée correcte des logs depuis les pods en service.