Quân Huỳnh Follow
Jul 4 · 7 min read · Listen

Setup Microservices on Kubernetes — Write a Configuration File

Deployed the microservice to Kubernetes

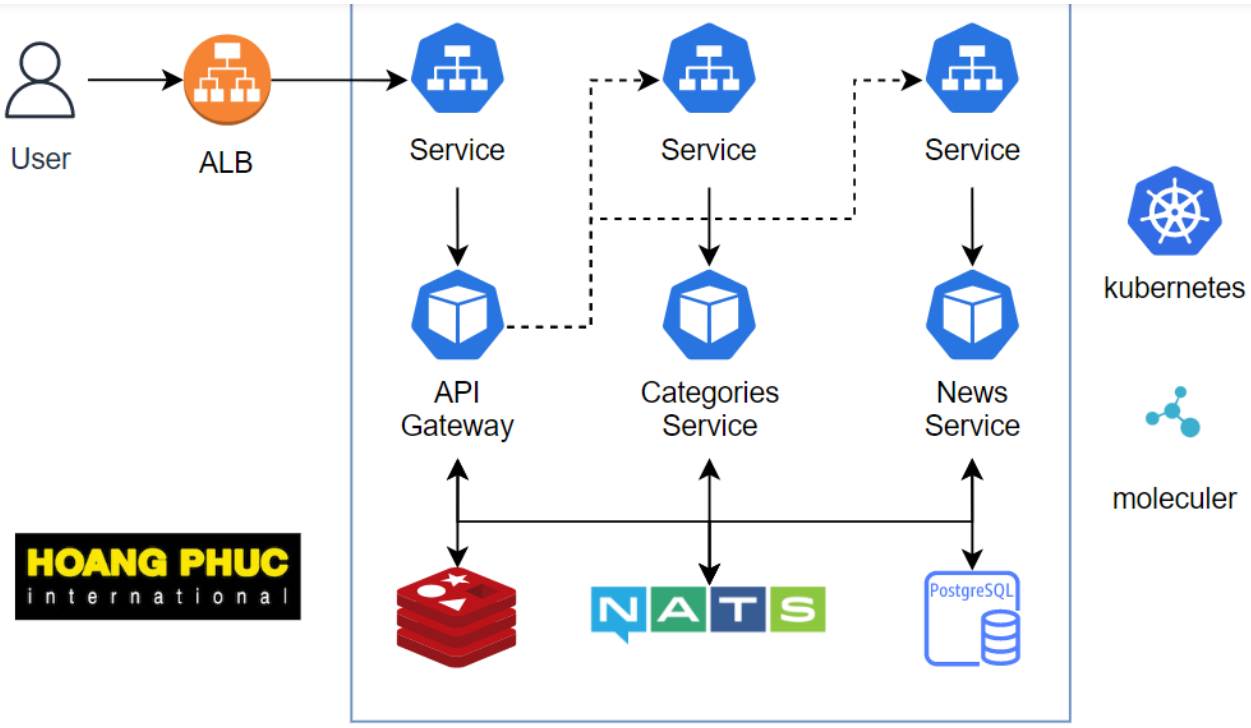
Photo by Savannah Wakefield on Unsplash

In this tutorial, we will learn how to set up a Microservices System on Kubernetes. In part one we are going to talk about how to write a configuration file for each component of the microservice system.

This is part one in the series Set up Microservice on Kubernetes:

1. Set up a Microservices on Kubernetes - Write Config File.
2. Set up a Microservices on Kubernetes - Automating Kubernetes with ArgoCD.
3. Set up a Microservices on Kubernetes - Implement CI/CD.

System Architecture



We use a framework called Moleculer to build a microservices system in the above diagram. Moleculer is a fast, modern, and powerful microservices framework for Node.js. It helps you to build efficient, reliable & scalable services. Moleculer provides many features for building and managing your microservices.

API Gateway exposes Moleculer services to end-users. The gateway is a regular Moleculer service running a (HTTP, WebSockets, etc.) server.

NATS, which is the transporter, it's a communication bus that services use to exchange messages. It transfers events, requests, and responses.

Categories and News services are a simple JavaScript module containing some part of a complex application. It is isolated and self-contained.

Cache System uses Redis and Database uses Postgres.

I have briefly talked about the architecture that we will deploy on Kubernetes, we will start working now.

Building Docker Container Image

Clone source code from https://github.com/hoalongnatsu/microservices. Go to the folder `microservices/code` and run the following commands to build an image. **The image name should be `<docker-hub-username>/microservice`.**

```
git clone https://github.com/hoalongnatsu/microservices.git && cd microservices/code
docker build . -t 080196/microservice
docker push 080196/microservice
```

Next, we are going to write a configuration file for each component.

Deploy API Gateway

First, we write a config file for API Gateway. Create a file named `api-gateway-deployment.yaml`.

```
apiVersion: apps/v1
```

```
spec:
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      component: api-gateway
  template:
    metadata:
      labels:
        component: api-gateway
    spec:
      containers:
        - name: api-gateway
          image: 080196/microservice
          ports:
            - name: http
              containerPort: 3000
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /
              port: http
          readinessProbe:
            httpGet:
              path: /
              port: http
          env:
            - name: NODE_ENV
              value: testing
            - name: SERVICEDIR
              value: dist/services
            - name: SERVICES
              value: api
            - name: PORT
              value: "3000"
            - name: CACHER
              value: redis://redis:6379
            - name: DB_HOST
              value: postgres
            - name: DB_PORT
              value: "5432"
            - name: DB_NAME
              value: postgres
            - name: DB_USER
              value: postgres
            - name: DB_PASSWORD
              value: postgres
            - name: TRANSPORTER
              value: nats://nats:4222
```

The image named `080196/microservice`, that we have previously built, including three services named `api`, `categories`, and `news`. We select the service that needs to run by passing the name of the service into an environment variable named SERVICES.

In the above config file, we pass the value as `api` for API gateway.

If you look at the code in the file code/services/api.service.ts, we will see the setting for the API gateway on line 15.

```
...
settings: {
  port: process.env.PORT || 3001,
...
```

With the PORT env, the API gateway listens on port 3000. The CACHER env use to declare the Redis host that the service uses. The env with prefix DB_ is used for Database. Run the following command to create Deployment.

```
kubectl apply -f api-gateway-deployment.yaml
```




```
api-gateway  0/1     1            0           100s
```

We have created the API Gateway, but when you get the pod. You will see that it does not run successfully but will be restarted over and over again.

```
$ kubectl get pod
NAME                           READY   STATUS    RESTARTS   AGE
api-gateway-79688cf6f5-g88f2   0/1     Running   2          93s
```

Check logs to find out why.

```
$ kubectl logs api-gateway-79688cf6f5-g88f2
...
[2021-11-07T14:53:37.449Z] ERROR api-gateway-79688cf6f5-g88f2-28/CACHER: Error: getaddrinfo EAI_AGAIN
redis
    at GetAddrInfoReqWrap.onlookup [as oncomplete] (dns.js:60:26) {
  errno: 'EAI_AGAIN',
  code: 'EAI_AGAIN',
  syscall: 'getaddrinfo',
  hostname: 'redis'
}
```

The error shown here is a pod cannot connect to Redis, because we have not created any Redis yet, next we will create Redis.

Deploy Redis

Create a file named `redis-deployment.yaml`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
  labels:
    component: redis
spec:
  strategy:
    type: Recreate
  selector:
    matchLabels:
      component: redis
  template:
    metadata:
      labels:
        component: redis
    spec:
      containers:
        - name: redis
          image: redis
          ports:
            - containerPort: 6379
```

Run the following commands.

```
$ kubectl apply -f redis-deployment.yaml
deployment.apps/redis created

$ kubectl get deploy
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
api-gateway   0/1     1            0           16m
redis         1/1     1            1           14s
```

```
  metadata:
    name: redis
    labels:
      component: redis
  spec:
    selector:
      component: redis
    ports:
      - port: 6379
```

Running a command.

```
kubectl apply -f redis-service.yaml
```

Restarting the API Gateway Deployment.

```
kubectl rollout restart deploy api-gateway
```

Check logs of the API Gateway and we still see it's not running.

```
$ kubectl logs api-gateway-7f4d5f54f-lzgkd
...
[2021-11-07T15:05:10.388Z] INFO  api-gateway-7f4d5f54f-lzgkd-28/CACHER: Redis cacher connected.

Sequelize CLI [Node: 12.13.0, CLI: 6.2.0, ORM: 6.6.5]

Loaded configuration file "migrate/config.js".
Using environment "testing".

ERROR: connect ECONNREFUSED 127.0.0.1:5432

Error: Command failed: sequelize-cli db:migrate
ERROR: connect ECONNREFUSED 127.0.0.1:5432
    at ChildProcess.exithandler (child_process.js:295:12)
    at ChildProcess.emit (events.js:210:5)
    at maybeClose (internal/child_process.js:1021:16)
    at Process.ChildProcess._handle.onexit (internal/child_process.js:283:5) {
  killed: false,
  code: 1,
  signal: null,
  cmd: 'sequelize-cli db:migrate'
}
Sequelize CLI [Node: 12.13.0, CLI: 6.2.0, ORM: 6.6.5]

Loaded configuration file "migrate/config.js".
Using environment "testing".

ERROR: connect ECONNREFUSED 127.0.0.1:5432
...
```

The error shown is a pod can not connect to the Database. Next, we will create a Database.

Deploy database

To deploy the Database, we use StatefulSet. Create a file named `postgres-statefulset.yaml`.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
  labels:
```

```
  template:
    metadata:
      labels:
        component: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:11
          ports:
            - containerPort: 5432
          volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgres-data
          env:
            - name: POSTGRES_DB
              value: postgres
            - name: POSTGRES_USER
              value: postgres
            - name: POSTGRES_PASSWORD
              value: postgres
  volumeClaimTemplates:
    - metadata:
        name: postgres-data
      spec:
        accessModes:
          - ReadWriteOnce
        storageClassName: hostpath
        resources:
          requests:
            storage: 5Gi
```

The `storageClassName` field depends on your Kubernetes cluster, you will specify the corresponding `storageClassName` field.
Run the following command to create STS.

```
kubectl apply -f postgres-statefulset.yaml
```

Create a file named `postgres-service.yaml` for the Database Service resources.

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
  labels:
    component: postgres
spec:
  selector:
    component: postgres
  ports:
    - port: 5432
```

Create it.

```
kubectl apply -f postgres-service.yaml
```

Restarting the API Gateway Deployment.

```
kubectl rollout restart deploy api-gateway
```

Check logs of the API Gateway and we will see it's running successfully.




```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: categories-service
  labels:
    component: categories-service
spec:
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      component: categories-service
  template:
    metadata:
      labels:
        component: categories-service
    spec:
      containers:
        - name: categories-service
          image: 080196/microservice
          env:
            - name: NODE_ENV
              value: testing
            - name: SERVICEDIR
              value: dist/services
            - name: SERVICES
              value: categories
            - name: CACHER
              value: redis://redis:6379
            - name: DB_HOST
              value: postgres
            - name: DB_PORT
              value: "5432"
            - name: DB_NAME
              value: postgres
            - name: DB_USER
              value: postgres
            - name: DB_PASSWORD
              value: postgres
            - name: TRANSPORTER
              value: nats://nats:4222

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: news-service
  labels:
    component: news-service
spec:
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      component: news-service
  template:
    metadata:
      labels:
        component: news-service
    spec:
      containers:
        - name: news-service
          image: 080196/microservice
          env:
            - name: NODE_ENV
              value: testing
            - name: SERVICEDIR
              value: dist/services
            - name: SERVICES
              value: news
            - name: CACHER
              value: redis://redis:6379
            - name: DB_HOST
              value: postgres
            - name: DB_PORT
              value: "5432"
```

```
            - name: TRANSPORTER
              value: nats://nats:4222
```

Create it.

```
kubectl apply -f categories-news-deployment.yaml
```

Next, we create a NATS transporter for our `moleculer` services can communicate with others.

Deploy NATS

Create a file named `nats-deployment.yaml`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nats
  labels:
    component: nats
spec:
  strategy:
    type: Recreate
  selector:
    matchLabels:
      component: nats
  template:
    metadata:
      labels:
        component: nats
    spec:
      containers:
        - name: nats
          image: nats
          ports:
            - containerPort: 4222
```

Create it.

```
kubectl apply -f nats-deployment.yaml
```

Create a file named `nats-service.yaml` for NATS.

```
apiVersion: v1
kind: Service
metadata:
  name: nats
  labels:
    component: nats
spec:
  selector:
    component: nats
  ports:
    - port: 4222
```

Create it.

```
kubectl apply -f nats-service.yaml
```

So our application has run successfully 😁.

But did you notice that the env variables we declare are a bit long and repetitive in our deployment files? We can make it clearer.

General Configuration Declaration

We can use ConfigMap for centralized configuration. Create a file named `microservice-cm.yaml`.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: microservice-cm
  labels:
    component: microservice-cm
data:
  NODE_ENV: testing
  SERVICEDIR: dist/services
  TRANSPORTER: nats://nats:4222
  CACHER: redis://redis:6379
  DB_NAME: postgres
  DB_HOST: postgres
  DB_USER: postgres
  DB_PASSWORD: postgres
  DB_PORT: "5432"
```

Create it.

```
kubectl apply -f microservice-cm.yaml
```

Update file `api-gateway-deployment.yaml`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-gateway
  labels:
    component: api-gateway
spec:
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      component: api-gateway
  template:
    metadata:
      labels:
        component: api-gateway
    spec:
      containers:
        - name: api-gateway
          image: 080196/microservice
          ports:
            - name: http
              containerPort: 3000
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /
              port: http
          readinessProbe:
            httpGet:
              path: /
              port: http
          env:
            - name: SERVICES
              value: api
            - name: PORT
              value: "3000"
```

Update file `categories-news-deployment.yaml`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: categories-service
  labels:
    component: categories-service
spec:
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      component: categories-service
  template:
    metadata:
      labels:
        component: categories-service
    spec:
      containers:
        - name: categories-service
          image: 080196/microservice
          env:
            - name: SERVICES
              value: categories
          envFrom:
            - configMapRef:
                name: microservice-cm

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: news-service
  labels:
    component: news-service
spec:
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      component: news-service
  template:
    metadata:
      labels:
        component: news-service
    spec:
      containers:
        - name: news-service
          image: 080196/microservice
          env:
            - name: SERVICES
              value: news
          envFrom:
            - configMapRef:
                name: microservice-cm
```

Update it.

```
kubectl apply -f api-gateway-deployment.yaml
kubectl apply -f categories-news-deployment.yaml
```

Check our system.

```
$ kubectl get pod
NAME                                READY   STATUS    RESTARTS
api-gateway-86b67895fd-cphmv        1/1     Running   0
categories-service-84c74cd87c-zjtd2   1/1     Running   0
```


So we have deployed the microservice to Kubernetes, as you can see, it's not difficult, is it?