# Node.js Application with ArgoCD

Fatih Yigit
ftygt29@gmail.com

## Introduction

In this project we are going to create whole ci/cd lifecycle of an application. The application will be a *Node.js* API which has a *Postgre SQL* db integration. We are going to create a helm chart for this application, and automate the CD pipeline with ArgoCD.

This project also includes *external secrets* and *Slack notifications* for *ArgoCD* events.

Whole architecture of the project:
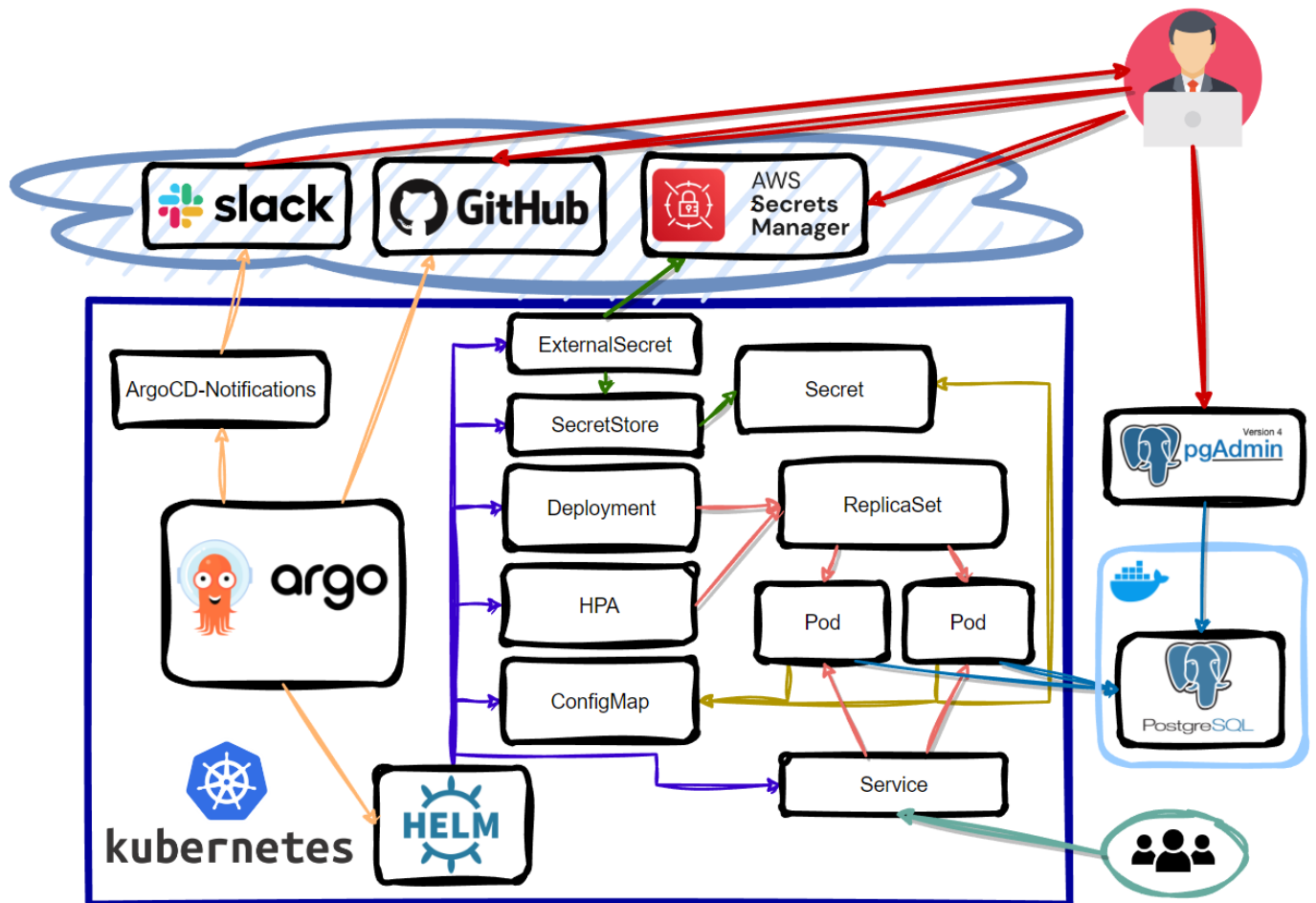


You should provide these prerequisites before follow the instructions:

- Docker or Docker Desktop
- Minikube
- Helm

This project have been studied on a Windows host, but the most steps will be similar with MacOS or Linux releases.

This documentation will be detailed, if you want to set up the whole environment quicker:

https://github.com/yigitf/nodeapp/blob/main/Readme.md

# Dockerizing the Application

The sample application that I used is:

https://github.com/isNan909/Node_API

I inspected the application to make sure how it works and what do we need:

```
const isProduction = process.env.NODE_ENV === 'production'


const connectionString =
`postgresql://${process.env.DB_USER}:${process.env.DATABASE_PASSWORD}@${process.env.DB_HOST}:${process.env.DB_PORT}/${process.env.DB_DATABASE}`


const pool = new Pool({
  connectionString: isProduction ? process.env.DATABASE_URL : connectionString,
  ssl: isProduction,
})
```

For development, everythings fine, but I do not have a real production PostgreSQL database for ssl connection. So, I have added a new environment variable that specifies if this db connection requires ssl or not.

```
const pool = new Pool({
  connectionString: isProduction ? process.env.DATABASE_URL : connectionString,
  ssl: process.env.NODE_SSL || false,
})
```

If we do not specify this *NODE_SSL* variable, it will not require a db connection via SSL.

Now, the main difference between production and development environments is production requires the connection string in a *single environment variable* for db connection; while development requires *5 separate environment variables* to create a connection string. This actually helps to differentiate and test these two environments.

To dockerize an application we should summarize the build process of the application we want to dockerize. This NodeJS application needs two steps "npm install" and "npm start". And we need a base image.

```
# a lightweight base image having necessary dependencies for a node application
FROM node:lts-alpine
# setting a directory to insert the application
WORKDIR /app
# copy package.json and package-lock.json and install dependencies
COPY package*.json ./
RUN npm install
# copy remaining and start the application
COPY . .
CMD ["npm", "start"]
```

Now, we can create the image with the following command:

```
 PS C:\Users\yati\Desktop\docker\kind\Node_API> docker build --no-cache -t yigitfa/sample-node:1.0.7 .
 [+] Building 6.3s (10/10) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 143B
 => [internal] load metadata for docker.io/library/node:lts-alpine
 => [1/5] FROM docker.io/library/node:lts-alpine@sha256:9e38d3d4117da74a643f67041c83914480b335c3bd44d37ccf5b5ad86cd715d1
 => [internal] load build context
 => => transferring context: 84.10kB
 => CACHED [2/5] WORKDIR /app
 => [3/5] COPY package*.json ./
 => [4/5] RUN npm install
 => [5/5] COPY . .
 => exporting to image
 => => exporting layers
 => => writing image sha256:d5e7c02838c2fab34e88f3fcab318bc55afb39ee85bb0e79d16b15cb89f22312
 => => naming to docker.io/yigitfa/sample-node:1.0.7
```

Build process follows the steps defined in the dockerfile. The reason I named the image this way is because I want to push it to my repository on Docker Hub.

```
● PS C:\Users\yati\Desktop\docker\kind\Node_API>  docker push yigitfa/sample-node:1.0.7
 The push refers to repository [docker.io/yigitfa/sample-node]
 dd709911c5ff: Pushed
 46ff075983d4: Pushed
 0e674c0d5476: Pushed
 6dd932b910c2: Layer already exists
 b256f2c132d9: Layer already exists
 6d4bec181d96: Layer already exists
 63247acccce3: Layer already exists
 5af4f8f59b76: Layer already exists
 1.0.7: digest: sha256:9b93af4d32d1a63603005666e182dbc379325eedebc96a6aed30061a47e0ac76 size: 1995
```

The existing layers are from my previous tests, Docker Hub does not want to pull the same data repeatedly.

# Installing the Database and Testing

The image has been created, but we cannot test the application without a DB.

The application has a PostgreSQL integration, I will simply create it as a docker container with simple credentials. Here is the docker-compose configuration:

```yaml
version: '3.8'
services:
  postgresdb:
    image: postgres
    restart: unless-stopped
    environment:
      POSTGRES_USER: dbuser
      POSTGRES_PASSWORD: password123
      POSTGRES_DB: appdatabase
    ports:
      - 2345:5432
    volumes:
      - db:/var/lib/postgres
volumes:
  db:
```

With the command "docker-compose up -d" it gets ready in a docker container. Now we need to create a database and generate some data for our sample application. The queries for this need is provided in the applications github repo, also in this repo "docker/postgres/food.sql":

```sql
CREATE DATABASE foodie;
-- /l foodie
-- /c foodie

CREATE ROLE api_user WITH LOGIN PASSWORD 'root';
ALTER ROLE api_user CREATEDB;

-- \q
psql -d postgres -U api_user;

CREATE TABLE food (
    ID SERIAL PRIMARY KEY,
    Dish VARCHAR(30) NOT NULL,
    Country VARCHAR(30) NOT NULL
);

INSERT INTO food (dish, country) VALUES  ('Migas', 'Mexican'),('Tom Yam', 'Thai');

SELECT * FROM food
```

I installed pgAdmin desktop application and connect the database with the connection string below:

"postgresql://dbuser:password123@192.168.0.18:2345/appdatabase" (192.168.0.18 is my desktop local IP address, you should check for your own environment, as it changes for different pcs or servers)

After configuring the database with these queries, the node.js application should work fine.

We can add the image that we created to the docker-compose.yaml file and test both database and nodejs API.

```yaml
version: '3.8'
services:
  postgresdb:
    image: postgres
    restart: unless-stopped
    environment:
      POSTGRES_USER: dbuser
      POSTGRES_PASSWORD: password123
      POSTGRES_DB: appdatabase
    ports:
      - 2345:5432
    volumes:
      - db:/var/lib/postgres
  app:
    depends_on:
      - postgresdb
    image: yigitfa/sample-node:1.0.7
    restart: unless-stopped
    ports:
      - 7050:3055
    environment:
      NODE_ENV: production
      DB_HOST: postgresdb
      DB_USER: dbuser
      DATABASE_PASSWORD: password123
      DB_DATABASE: appdatabase
      DB_PORT: '5432'
      PORT: 3055
volumes:
  db:
```

You may notice that the port and hostname is different from the variables we previously used to connect the db.

In this scenario, the application and db is in the same compose and same local area network. So, they can reach each other with their hostnames and internal port numbers that the container exposes.

After applying new docker compose file we can test the application with "/food" endpoint.

```
PS C:\Users\yati\Desktop\docker\kind\Node_API> curl http://localhost:7050/food


StatusCode      : 200
StatusDescription : OK
Content         : {"status":"sucess","data":[{"id":1,"dish":"Migas","country":"Mexican"},{"id":2,"dish":"
Tom Yam","country":"Thai"}]}
RawContent      : HTTP/1.1 200 OK
                  X-DNS-Prefetch-Control: off
                  X-Frame-Options: SAMEORIGIN
                  Strict-Transport-Security: max-age=15552000; includeSubDomains
                  X-Download-Options: noopen
                  X-Content-Type-Options: nosniff
```

Success! We can see the data that we generated, it shows that the application connects the database without an issue.

# Creating the Kubernetes Cluster and Manifests

Everything is fine, but we may serve this application for millions of users and may want to automate ci/cd lifecycle or autoscale the pods and orchestrate the containers. This is a demo project for Kubernetes container orchestration tool, so we need a Kubernetes cluster.

A single node Kubernetes Cluster with (*3CPUs* and *6GB Memory*) can be created with:

```
PS C:\Users\yati\Desktop\docker\dev\nodeapp> minikube start --memory 6000 --cpus 3
😄  minikube v1.32.0 on Microsoft Windows 11 Pro 10.0.22621.2861 Build 22621.2861
✨  Automatically selected the docker driver. Other choices: virtualbox, ssh
📌  Using Docker Desktop driver with root privileges
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🔥  Creating docker container (CPUs=3, Memory=6000MB) ...
🐳  Preparing Kubernetes v1.28.3 on Docker 24.0.7 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔗  Configuring bridge CNI (Container Networking Interface) ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

I have created a configmap for the environment variables to connect the database.

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: configmap1
data:
  NODE_ENV: development
  DB_HOST: "192.168.0.18"
  DB_USER: dbuser
  DATABASE_PASSWORD: password123
  DB_DATABASE: appdatabase
  DB_PORT: "2345"
  PORT: "3055"
```

Even if the Kubernetes Cluster is also runs on Docker, they are in separate networks with Postgre SQL container. So, the application in the cluster should connect the database from desktops IP address. That's why we have IP and external port of the container in this ConfigMap.

We need a deployment manifest to create a ReplicaSet and coordinate the pods:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodeapp
  labels:
    app: nodeapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nodeapp
  template:
    metadata:
      labels:
        app: nodeapp
    spec:
      containers:
      - name: yigitfcontainer
        image: yigitfa/sample-node:1.0.6
        ports:
        - containerPort: 3055
        envFrom:
        - configMapRef:
            name: configmap1
```

It will create 3 pods and the pods will take the environment variables from the ConfigMap that we just created.

And these pods will need a service that exposes the application pods and does the loadbalancing.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nodeapp
  labels:
    app: nodeapp
spec:
  ports:
  - port: 3055
    targetPort: 3055
  selector:
    app: nodeapp
```

To create these manifests:

```
PS C:\Users\yati\Desktop\docker\nodeapp\manifests> kubectl apply -f .\configmap.yaml
 -f .\deployment.yaml -f .\service.yaml
configmap/configmap1 unchanged
deployment.apps/nodeapp created
service/nodeapp created
```

Manifests are created, now we can see what's inside of the cluster.

```
PS C:\Users\yati\Desktop\docker\nodeapp\manifests> kubectl get all
NAME                              READY   STATUS    RESTARTS   AGE
pod/nodeapp-f6c945565-8kpnp       1/1     Running   0          19s
pod/nodeapp-f6c945565-lrpg4       1/1     Running   0          19s
pod/nodeapp-f6c945565-szzkc       1/1     Running   0          19s

NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/kubernetes    ClusterIP   10.96.0.1        <none>        443/TCP    12m
service/nodeapp       ClusterIP   10.102.180.176   <none>        3051/TCP   19s

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nodeapp     3/3     3            3           19s

NAME                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/nodeapp-f6c945565      3         3         3       19s
```

We have 3 pods, 2 services(one of them is the default service that minikube creates), our deployment and the replicaset created by the deployment.

The service I created exposes port 3051, I will forward it to the outside in another terminal as this command will be attached on the terminal.

```
PS C:\Users> kubectl port-forward svc/nodeapp 8081:3051
Forwarding from 127.0.0.1:8081 -> 3051
Forwarding from [::1]:8081 -> 3051
```

```
PS C:\Users\yati\Desktop\docker\nodeapp\manifests> curl http://localhost:8081/food


StatusCode        : 200
StatusDescription : OK
Content           : {"status":"sucess","data":[{"id":1,"dish":"Migas","country":"Me
                    xican"},{"id":2,"dish":"Tom Yam","country":"Thai"}]}
RawContent        : HTTP/1.1 200 OK
                    X-DNS-Prefetch-Control: off
                    X-Frame-Options: SAMEORIGIN
                    Strict-Transport-Security: max-age=15552000; includeSubDomains
                    X-Download-Options: noopen
                    X-Content-Type-Options: nosniff
```

This shows that the tests on these manifests are successful.

# Environment Management and External Secrets

Until now, we just built "development" environments on our tests, however there will be one more environment which is called "production". As we discussed earlier, there are differences between prod and dev, they use different environment variables for database connection.

I am going to create one more difference, I will use *external secrets* for secret management in prod environment. Actually, we have one secret variable for production which is *DATABASE_URL* (the connection string for PostgreSQL DB).

I used AWS Secrets Manager to manage my external secret.

| Gizli bilgi anahtarı | Gizli bilgi değeri |
| --- | --- |
| db_url | postgresql://dbuser:password123@192.168.0.18:2345/appdatabase |

After creating the secret, I generated a user and an access keys for this user with the permission below on the *AWS IAM* panel.

| | Policy name [⤢] | ▲ | Type |
|---|---|---|---|
| ☐ | ⊞ 📦 SecretsManagerReadWrite | | AWS managed |

I will use these *access key* and *secret access key* to access my external secret from the Kubernetes cluster. First, we need to encode these keys, encoding methods depends on your operating system, so it is also safe to use "www.base64encode.org" website.

After encoding, we can create a secret file for aws authentication.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: awssm-secret
data:
  access-key: QUtJQVJOQkQ0SFNNWFpWU1JJJM0w=
  secret-access-key: azlCS0NmTGgwcmdUN3ZaRTZ0M29WeFd6amxNcytCU290NHhISkdzaQ==
type: Opaque
```

It is also required to create a *secret store* and an *external secret* for pulling the secret key from aws. But first we need to install the dependencies with:

```
helm repo add external-secrets https://charts.external-secrets.io
helm install external-secrets external-secrets/external-secrets -n external-secrets --create-namespace
```

```
NAME: external-secrets
LAST DEPLOYED: Sun Dec 24 14:53:20 2023
NAMESPACE: external-secrets
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
external-secrets has been deployed successfully!
```

```yaml
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: secretstore-aws
spec:
  provider:
    aws:
      service: SecretsManager
      region: eu-north-1
      auth:
        secretRef:
          accessKeyIDSecretRef:
            name: awssm-secret
            key: access-key
          secretAccessKeySecretRef:
            name: awssm-secret
            key: secret-access-key
---
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
```

```
    name: externalsecret-aws
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: secretstore-aws
    kind: SecretStore
  target:
    name: external-secret-prod
    creationPolicy: Owner
  data:
  - secretKey: DATABASE_URL
    remoteRef:
      key: postgre
      property: db_url
```

In secret store manifest there are authentication configurations, it also uses access keys that we created.

On the other hand, in the external secret template we configure which key to pull and which secret inside the cluster will be overwrited or generated.

For development environment I will also use *secrets* for *database credentials*, environment and application port variables will be stored on separate *configmaps* for production and environment.

Again, as we discussed above; the main difference between production and development environments is production requires the connection string in a *single environment variable* for db connection; while development requires *5 separate environment variables* as credentials to create a connection string.

The environment management architecture will look like the following diagram.
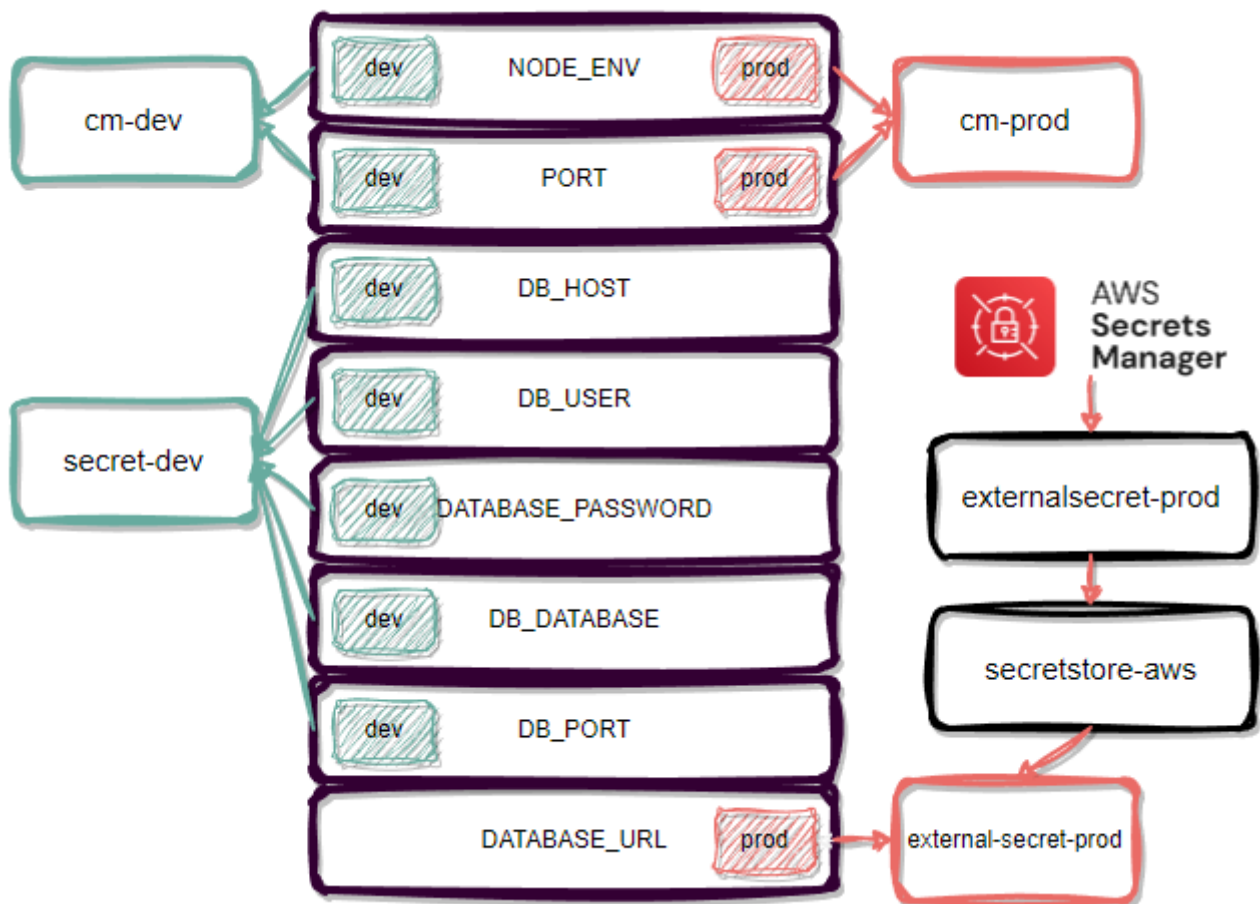
# Creating the Helm Chart

The application and environment management have been designed. Manifests are working just fine. So, it is time to create a Helm chart for the application.

Helm automates the creation, configuration and packaging the Kubernetes manifests. We can collect all dependent manifests together, configure and build them together.

To create a simple helm chart:

```
PS C:\Users\yati\Desktop\docker\deneme> helm create helm-node
Creating helm-node
```

We will not need any of these templates; so, I will simply delete them and create our own templates.

Before deployment and other manifests, I am going to create configmaps and secrets.
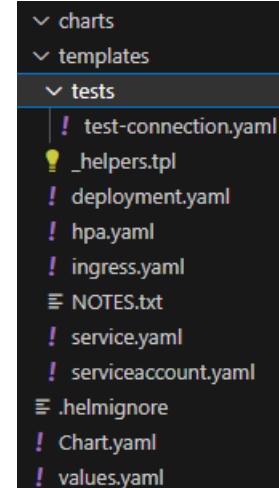
Actually, there is nothing to change for *external secret* management, we will just include these three manifests to our helm chart.

*Configmaps* will be simple and the variable in the *secret* file for development environment will be encoded on base64 as usual.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: cm-dev
data:
  NODE_ENV: development
  PORT: "8080"
---
kind: Secret
apiVersion: v1
metadata:
  name: secret-dev
data:
  DB_HOST: MTkyLjE2OC4wLjE4
  DB_USER: ZGJ1c2Vy
  DATABASE_PASSWORD: cGFzc3dvcmQxMjM=
  DB_DATABASE: YXBwZGF0YWJhc2U=
  DB_PORT: MjM0NQ==
type: Opaque
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: cm-prod
data:
  NODE_ENV: production
  PORT: "8090"
```

Now it is time to create a *values.yaml* file. I want to run *two different environments* with this *single helm chart*, so I should specify *environment, service port number*, *configmap* and *secret* objects from the values file.

The values file will also include some dynamic variables like *app name, replica count, image name and tag*, *port number* and *pod resources* which we may want to change or update.

```
appName: sample-node-app
```

```yaml
image: yigitfa/sample-node:1.0.6
environment: production
replicas: 2
port: 8090 # should be the same with environment
configmap:
  name: cm-prod
secrets:
  name: external-secret-prod
resources:
  requests:
    memory: "16Mi"
    cpu: "50m"
  limits:
    memory: "128Mi"
    cpu: "100m"
```

Now, we can create *deployment* and *ClusterIP service* manifests with these variables.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.appName}}
  labels:
    app: {{ .Values.appName}}
spec:
  replicas: {{ .Values.replicas}}
  selector:
    matchLabels:
      app: {{ .Values.appName}}
  template:
    metadata:
      labels:
        app: {{ .Values.appName}}
    spec:
      containers:
      - name: {{ .Values.appName}}
        image: {{ .Values.image}}
        ports:
        - containerPort: {{ .Values.port}}
        envFrom:
        - configMapRef:
            name: {{ .Values.configmap.name}}
        - secretRef:
            name: {{ .Values.secrets.name}}
        resources:
          requests:
            memory: {{ .Values.resources.requests.memory}}
            cpu: {{ .Values.resources.requests.cpu}}
          limits:
            memory: {{ .Values.resources.limits.memory}}
            cpu: {{ .Values.resources.limits.cpu}}
---
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.appName}}
  namespace: {{ .Values.namespace}}
  labels:
    app: {{ .Values.appName}}
spec:
  ports:
```

```
    - port: {{ .Values.port}}
      targetPort: {{ .Values.port}}
  selector:
    app: {{ .Values.appName}}
```

I will also add a *Horizontal Pod Autoscaler* to this chart to autoscale our pods depend on their cpu usage.

We need to add some dynamic variables to the *values.yaml* file for our HPA:

```
hpa:
  minreplicas: 1
  maxreplicas: 7
  cputhreshold: 50
```

Now, we can create the autoscaler with these values.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ .Values.appName}}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ .Values.appName}}
  minReplicas: {{ .Values.hpa.minreplicas}}
  maxReplicas: {{ .Values.hpa.maxreplicas}}
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: {{ .Values.hpa.cputhreshold}}
```

There is one more thing for HPA, it needs monitoring data to take action on the right time. Install the metrics-server with the command below:

```
# minikube addons enable metrics-server
```

The autoscaler will scale the pods in the range of 1-7 replicas according to cpu usage with these values.

```
PS C:\Users\yati\Desktop\docker\nodeapp\helm> kubectl create namespace helm-node
namespace/helm-node created
PS C:\Users\yati\Desktop\docker\nodeapp\helm> helm install -n helm-node helm-node
NAME: helm-node
LAST DEPLOYED: Sun Dec 24 15:03:02 2023
NAMESPACE: helm-node
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Now, we have installed the helm chart.

```
PS C:\Users\yati\Desktop\docker\nodeapp\helm> helm list --all-namespaces
NAME                    NAMESPACE             REVISION    UPDATED
                        STATUS        CHART                 APP VERSION
external-secrets        external-secrets      1           2023-12-24 14:53:20.0520941 +0300 +03    deployed    external-secrets-0.9.10 v0.9.10
helm-node               helm-node           | 1           2023-12-24 15:03:02.5502704 +0300 +03    deployed    helm-node-0.1.0        1.16.0
```

We can see our helm chart and the chart that we installed for external secret management.
Anyways, we can see what did we just created in helm-node namespace.

```
PS C:\Users\yati\Desktop\docker\nodeapp\helm> kubectl get all -n helm-node
NAME                                     READY   STATUS    RESTARTS   AGE
pod/sample-node-app-65ccbc7b74-stmkt     1/1     Running   0          9m20s
pod/sample-node-app-65ccbc7b74-t5mmv     1/1     Running   0          9m20s

NAME                      TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/sample-node-app   ClusterIP   10.103.6.182   <none>        8090/TCP   9m20s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/sample-node-app     2/2     2            2           9m20s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/sample-node-app-65ccbc7b74    2         2         2       9m20s

NAME                                                           REFERENCE                   TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/sample-node-app           Deployment/sample-node-app  <unknown>/50%  1         7         2          9m20s
```

Everything is created as configured, so I will forward the service port and test the database connection.

```
PS C:\Users\yati> kubectl port-forward svc/sample-node-app -n helm-node 8090:8090
Forwarding from 127.0.0.1:8090 -> 8090
Forwarding from [::1]:8090 -> 8090
Handling connection for 8090
```

```
PS C:\Users\yati\Desktop\docker\nodeapp\helm> curl http://localhost:8090/food

StatusCode      : 200
StatusDescription : OK
Content         : {"status":"sucess","data":[{"id":1,"dish":"Migas","country":"Mexican"},{"id":2,"dish":"Tom Yam","country":"Thai"}]}
RawContent      : HTTP/1.1 200 OK
                  X-DNS-Prefetch-Control: off
                  X-Frame-Options: SAMEORIGIN
                  Strict-Transport-Security: max-age=15552000; includeSubDomains
                  X-Download-Options: noopen
                  X-Content-Type-Options: nosniff
```

It is working. The helm chart has been configured as *production* environment, so that means our *external secret operator* pulls the secret data from remote *aws secret manager* without an issue.

# Integrate the Helm Chart with ArgoCD

We have packaged all manifests and we are able to configure them with a simple values.yaml file. But it is not the best practice for managing applications on Kubernetes.

We can store this Helm chart on a Git repository and synchronize this repository with Kubernetes cluster thanks to *ArgoCD*. We can automate Kubernetes *CD lifecycle* and *monitor* it, also we can receive *notification* alerts with ArgoCD.

First, we need to install ArgoCD on our cluster with following command:

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml
```

To forward the web port of ArgoCD GUI run the following on a separate terminal.

```
PS C:\Users\yati> kubectl port-forward svc/argocd-server -n argocd 8070:443
Forwarding from 127.0.0.1:8070 -> 8080
Forwarding from [::1]:8070 -> 8080
```

GUI: http://localhost:8070

The username will be admin and we should decode the password on base64:

```
PS C:\Users\yati\Desktop\docker\nodeapp\helm> kubectl get secret argocd-initial-admin-secret -n argocd -o yaml
apiVersion: v1
data:
  password: VHJFdjlaQmV0djlyNHRxUA==
kind: Secret
```

After connecting to GUI, we are going to sync ArgoCD with a git repository, but do we have one yet?

I have created a repo with two extra branches: *dev* and *prod*.

Now, we can create *ArgoCD application* configurations for both "dev" and "prod" branches of this Github repository.

Here is the application configuration of *production* environment.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nodejs-prod
  namespace: argocd
spec:
  destination:
    namespace: prod  # desired namespace for this helm chart for this branch
    server: 'https://kubernetes.default.svc'
  source:
    path: helm
    repoURL: 'https://github.com/yigitf/nodeapp.git' # git repo that to synchronize
    targetRevision: prod # the branch that the application synchronize
  sources: []
  project: default
  syncPolicy:
    automated:
      prune: true # delete resources is no longer defined in repo
      selfHeal: true # automated synchronize with a timeout
    syncOptions:
      - CreateNamespace=true # create the namespace if does not exist
```
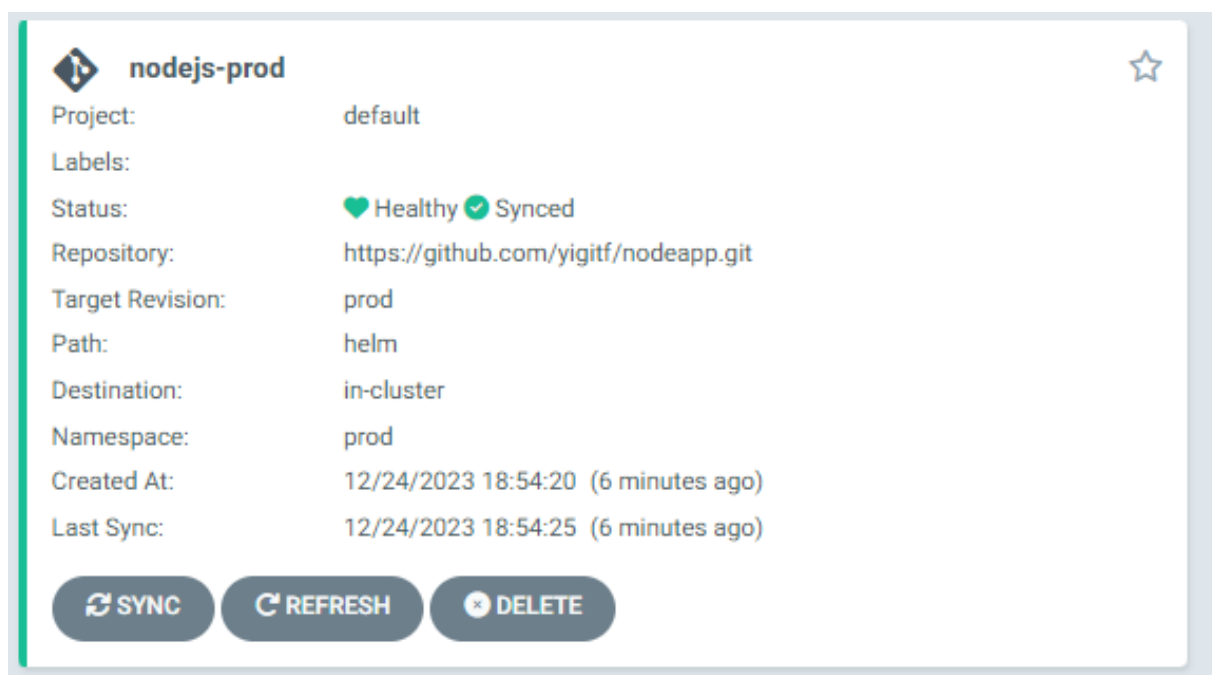
Application of the "development" environment will be similar, just replace all "prod" words with "dev".

You can create this environment with the application configuration in this repository.

https://github.com/yigitf/nodeapp/blob/main/argocd/application-prod.yaml

```
# kubectl apply –f application-prod.yaml
```

We can see the deployment on https://localhost:8070/applications you should see the application like below.



When I click this application, I can see the whole architecture:
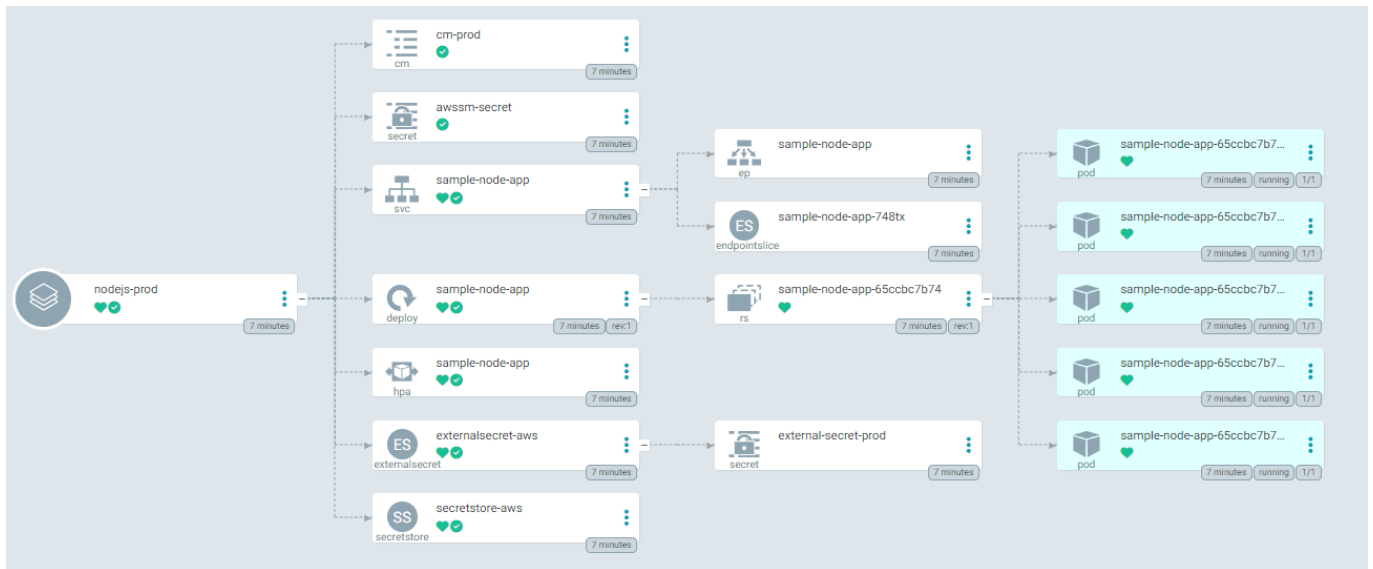
We can run these two applications at the same time, on different namespaces.

```
# kubectl apply -f application-dev.yaml
```



# ArgoCD Notifications and CD Tests

We have deployed our ArgoCD application, now it is time to test it. It is a good idea to test the application with notifications.

These commands below will install the argocd notifications.

```
# kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj-labs/argocd-notifications/release-1.0/manifests/install.yaml
# kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj-labs/argocd-notifications/release-1.0/catalog/install.yaml
```

```
PS C:\Users\yati\Desktop\docker\nodeapp\argocd> kubectl apply -n argocd -f https://raw.githubusercontent.com/a
rgoproj-labs/argocd-notifications/release-1.0/manifests/install.yaml
serviceaccount/argocd-notifications-controller configured
role.rbac.authorization.k8s.io/argocd-notifications-controller configured
rolebinding.rbac.authorization.k8s.io/argocd-notifications-controller configured
configmap/argocd-notifications-cm configured
secret/argocd-notifications-secret configured
service/argocd-notifications-controller-metrics configured
deployment.apps/argocd-notifications-controller configured
PS C:\Users\yati\Desktop\docker\nodeapp\argocd> kubectl apply -n argocd -f https://raw.githubusercontent.com/a
rgoproj-labs/argocd-notifications/release-1.0/catalog/install.yaml
configmap/argocd-notifications-cm configured
```

Now, we have installed the argocd-extension but we need to configure it.

I will connect ArgoCD Notifications to my *Slack* channel. I followed https://argocd-notifications.readthedocs.io/ documentations to authentication and configuration of notifications.

I have created an *api user* and generated a *token* on my *Slack account*.

## OAuth Tokens for Your Workspace

These tokens were automatically generated when you installe
You can use these to authenticate your app. Learn more.

**Bot User OAuth Token**

xoxb-6375366232327-6375372266935-ulZsnG1ReJR2vg

Access Level: Workspace

I inserted this token to "argocd-notification-secret" *without* encoding it.

```
apiVersion: v1
kind: Secret
metadata:
  name: argocd-notifications-secret
  namespace: argocd
stringData:
  slack-token: xoxb-6375366232327-6375372266935-ulZsnG1ReJR2vg*****************
```

Then I create the configurations in "argocd-notifications-cm" configMap.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-notifications-cm
  namespace: argocd
data:
  context: |
    argocdUrl: http://localhost:8080
  service.slack: |
    token: $slack-token
```

So far so good, we need triggers for our notifications. I will use sample templates on argocd-notifications documentation.

```
  template.app-sync-succeeded: |
    message: |
      {{if eq .serviceType "slack"}}:white_check_mark:{{end}} Application
{{.app.metadata.name}} has been successfully synced at
{{.app.status.operationState.finishedAt}}.
      Sync operation details are available at:
{{.context.argocdUrl}}/applications/{{.app.metadata.name}}?operation=true .
    slack:
      attachments: |
        [{
          "title": "{{ .app.metadata.name}}",
          "title_link":"{{.context.argocdUrl}}/applications/{{.app.metadata.name}}",
          "color": "#18be52",
          "fields": [
          {
            "title": "Sync Status",
            "value": "{{.app.status.sync.status}}",
```

```
          "short": true
        },
        {
          "title": "Repository",
          "value": "{{.app.spec.source.repoURL}}",
          "short": true
        }
        {{range $index, $c := .app.status.conditions}}
        {{if not $index}},{{end}}
        {{if $index}},{{end}}
        {
          "title": "{{$c.type}}",
          "value": "{{$c.message}}",
          "short": true
        }
        {{end}}
        ]
      }]
    deliveryPolicy: Post
    groupingKey: ""
    notifyBroadcast: false
  trigger.on-sync-succeeded: |
    - description: Application syncing has succeeded
      send:
      - app-sync-succeeded
      when: app.status.operationState.phase in ['Succeeded']
```

Now, I need to apply these manifests and specify the annotation in the ArgoCD application configurations.

```
annotations:
  notifications.argoproj.io/subscribe.on-sync-succeeded.slack: alerts
```

After applying and configuring the Application configuration, I have created a channel named "alerts" in my slack channel.

I have added one more notification which alerts when application is successfully up and running.

To test it, I just commited a new image version to the dev branch of my github repo.



Both ArgoCD synchronization and notifications seem working without an issue.

# CI Logic and Gitlab CI

In this project, we did all the CI operations manually and automated the CD operations. If we want to automate these CI operations too, what do we need to do? The short answer is, automate the manual operations.

The manual CI steps that I do for this project are:

- Git clone (sample node application)
- Run the application
- Test (a healthcheck for this example)
- Docker build –t image:tag
- Docker push
- Clone the helm repo
- Edit the image name or any desired variable in *values.yaml*
- Commit!
- <u>ArgoCD does the rest!</u>

I can simply divide these steps into three stages:

Test:

- Git clone (if the repo is stored on Gitlab; Gitlab-ci does not need to clone the repository again)
- Run the application
- Test (a healthcheck for this example)

Build:

- Docker build –t image:tag
- Docker push

Deploy: (Actually it only triggers the deployment)

- Clone the helm repo
- Edit the image name or any desired variable in values.yaml
- Commit!

I have created a CI pipeline for the production environment with *gitlab-ci.yaml:*

```yaml
stages:
  - test
  - build
  - deploy
variables:
  IMAGE_NAME: yigitfa/sample_node
  IMAGE_TAG: 1.0.8
  GITHUB_REPO: https://github.com/yigitf/nodeapp.git
  GITHUB_TOKEN: <github-token>

test:
  stage: test
  image: node:latest
  before_script:
    - npm install
    - npm start
  script:
    - curl http:/localhost:3000/food

build:
  stage: build
  image: docker:latest
  before_script:
    - docker login -u $REGISTRY_USER -p $REGISTRY_PASS
  script:
    - docker build -t $IMAGE_NAME:$IMAGE_TAG .
    - docker push $IMAGE_NAME:$IMAGE_TAG

deploy:
  stage: deploy
  script:
    - apt-get update && apt-get install -y git
    - git config --global user.email "fatihyigit29@hotmail.com"
    - git config --global user.name "yigitf"
    - git clone $GITHUB_REPO
    - cd nodeapp
    - git checkout prod
    - sed -i "s|image:..*|image: $IMAGE_NAME:$IMAGE_TAG|g" helm/values.yaml
    - git add helm/values.yaml
    - git commit -m "Update image in values.yaml to $IMAGE_NAME:$IMAGE_TAG"
    - git push origin prod
  only:
    - prod
```

And the complete CI/CD pipeline would look like the diagram below.