Local Kubernetes Cluster Deployment

Blind Date Platform

Date	**:**	**04.06.2023**
Version	**:**	**v1**
Status	**:**	**Complete**
Author	**:**	**Stanislav Petkov**

Table of Contents

Introduction

This guide provides step-by-step instructions on deploying and managing our project's backend on a local Kubernetes cluster. By following this guide, you'll be able to replicate the production environment on your machine for testing purposes, such as load testing, which can be costly in the cloud. This setup involves Docker, Minikube, and the Krakend API Gateway. **This guide assumes that you already have a production pipeline running. Open the directory `/k8s/local` to start.**

Step 1: Docker and Kubernetes Installation

This guide assumes that Docker is already installed on your machine. If not, please install Docker appropriate to your operating system.

Once Docker is ready, enable Kubernetes in Docker. For Docker Desktop (Windows/Mac), this is done through the Docker Desktop Dashboard:

1. Click the Docker icon in your taskbar.
2. Go to 'Settings'.
3. Click on the 'Kubernetes' tab.
4. Check 'Enable Kubernetes'.

For Docker in Linux, consider using Minikube (which we install in the next step) as both the Docker runtime and the single-node Kubernetes cluster.

Step 2: Minikube and kubectl Installation

Minikube is a tool that runs a single-node Kubernetes cluster on your personal computer. Install Minikube following the official Kubernetes guide. After installation, check it by running **`minikube version`**.

kubectl is a command-line tool for controlling Kubernetes clusters. Install it using package managers like Homebrew, Snap, or Chocolatey, depending on your operating system. For example, on MacOS, install it via Homebrew:

```
brew install kubectl
```

Verify the installation with `**kubectl version**`.

Step 3: Starting Minikube

Next start a Minikube cluster using Docker as the driver and assign it 4000MB of memory. This cluster will host your local Kubernetes environment.

```
minikube start --driver=docker --memory=4000
```

Step 4: Creating Docker Secret

Replace the placeholders `<docker-server>`, `<docker-username>`, `<docker-password>`, and `<docker-email>` in the command below with your actual details:

```
kubectl create secret docker-registry acr-credentials --docker-server=<docker-server> --docker-username=<docker-username> --docker-password=<docker-password> --docker-email=<docker-email>
```

This secret is used to access the Azure Container Registry (ACR) where your Docker images are stored.

Step 5: Setting up Nginx Ingress Controller

Deploy the Nginx Ingress Controller to manage inbound traffic routing and load balancing for our application using the command:

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.0.0/deploy/static/provider/cloud/deploy.yaml

This controller will expose our application locally for testing and development.

Step 6: Apply Kustomization

Update the image names and tags in kustomize.yaml with the new group's ACR info. This file is responsible for managing all your Kubernetes objects.

kubectl apply -k .

This command applies the kustomize.yaml configuration and deploys your applications (post-service, user-service, comment-service, krakend) onto the Kubernetes cluster.

Step 7: Port Forwarding

Replace <pod-name> in the command below with the actual pod name of your Nginx ingress controller.

kubectl port-forward <pod-name> 80:80 -n ingress-nginx

This command maps port 80 of your local machine to port 80 of the Ingress controller, allowing you to access the API at localhost:80. Remember to replace <pod-name> with the actual name of your pod. You can get the pod name by running kubectl get pods -n ingress-nginx. Finally, you can access the API at localhost:80.