

**Kubernetes installation and Configuration with multiple Container using Local Registry  
(Single Node Cluster)**

**Version History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SI | Date | Ver | Description | Changed by | Approved/Reviewed by |
| 01 | 07-Sep-21 | 1.0 | Kubernetes installation and Configuration in Local Environment using Local Registry | Maidul Islam  Software Engineer | Md. Mahedee Hasan  Head of Software Development |

Contents

[Introduction 2](#_Toc81675967)

[What Kubernetes Does? 2](#_Toc81675968)

[Docker Hub 3](#_Toc81675969)

[On the Need for Orchestration 3](#_Toc81675970)

[Kubernetes 3](#_Toc81675971)

[Kubernetes Basics 3](#_Toc81675972)

[Node 4](#_Toc81675973)

[Pods 4](#_Toc81675974)

[Deployment 4](#_Toc81675975)

[Vendor Agnostic 4](#_Toc81675976)

[Use Cases 5](#_Toc81675977)

[Kubernetes Architecture 5](#_Toc81675978)

[Master Node 5](#_Toc81675979)

[Worker/Slave nodes 6](#_Toc81675980)

[Virtual Kubelet (Nodeless Kubernetes) 6](#_Toc81675981)

[Step 1: Install Kubectl. 7](#_Toc81675982)

[Step 2: Install & Setup Hyper-V 7](#_Toc81675983)

[Step 3: Install Docker for Windows 8](#_Toc81675984)

[Step 4: Install Kubernetes on Windows 10 8](#_Toc81675985)

[Step 5: Create a Account in docker Hub 9](#_Toc81675986)

[Step 6: Create a .net core Project or Clone 9](#_Toc81675987)

[Step 7: Working with Pods 10](#_Toc81675988)

[Step 8: Kubernetes Service 12](#_Toc81675989)

[Step 9: Install Kubernetes Dashboard 13](#_Toc81675990)

[Step 10: Access the dashboard 13](#_Toc81675991)

[Deploy with multiple pod 17](#_Toc81675992)

[References 19](#_Toc81675993)

# Introduction

Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications. The open source project is hosted by the Cloud Native Computing Foundation (CNCF).

Kubernetes provides an easy way to scale an application, compared to virtual machines. It keeps code operational and speeds up the delivery process.

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

It is an open-source container orchestration platform that automates deployment, management and scaling of applications.

Kubernetes came out of google as an open-source project.

All big companies like Amazon, Microsoft, Google, IBM offers hosted Kubernetes services.

Kubernetes can be setup on premise as well.

# What Kubernetes Does?

Before we explain what Kubernetes does, we need to explain what containers are and why people are using those.

A container is a mini-virtual machine. It is small, as it does not have device drivers and all the other components of a regular virtual machine. Docker is by far the most popular container and it is written in Linux. Microsoft also has added containers to Windows as well, because they have become so popular.

The best way to illustrate why this is useful and important is to give an example.

Suppose you want to install the nginx web server on a Linux server. You have several ways to do that. First, you could install it directly on the physical server’s OS. But most people use virtual machines now, so you would probably install it there.

But setting up a virtual machine requires some administrative effort and cost as well. And machines will be underutilized if you just dedicate it for just one task, which is how people typically use VMs. It would be better to load that one machine up with nginx, messaging software, a DNS server, etc.

The people who invented containers thought through these issues and reasoned that since nginx or any other application just needs some bare minimum operating system to run, then why not make a stripped down version of an OS, put nginx inside, and run that. Then you have a self-contained, machine-agnostic unit that can be installed anywhere.

Now containers are so popular than they threaten to make VMs obsolete, is what some people say.

## Docker Hub

But making the container small is not the only advantage. The container can be deployed just like a VM template, meaning an application that is ready to go that requires little or no configuration.

There are thousands of preconfigured Docker images at the Dockerhub public repository. There, people have taken the time to assemble opensource software configurations that might take someone else hours or days to put together. People benefit from that because they can install nginx or even far more complicated items simply by downloading them from there.

For example, this one line command will down, install, and start Apache Spark with Jupyter notebooks (iPython):

**docker run -d -p 8888:8888 jupyter/all-spark-notebook**

As you can see it is running on port 8888. So you could install something else on another port or even install a second instance of Spark and Jupyter.

## On the Need for Orchestration

Now, there is an inherent problem with containers, just like there is with virtual machines. That is the need to keep track of them. When public cloud companies bill you for CPU time or storage then you need to make sure you do not have any orphaned machines spinning out there doing nothing. Plus there is the need to automatically spin up more when a machine needs more memory, CPU, or storage, as well as shut them down when the load lightens.

Orchestration tackles these problems. This is where Kubernetes comes in.

## Kubernetes

Google built Kubernetes and has been using it for 10 years. That it has been used to run Google’s massive systems for that long is one of its key selling points. Two years ago Google pushed Kubernetes into open source.

Kubernetes is a cluster and container management tool. It lets you deploy containers to clusters, meaning a network of virtual machines. It works with different containers, not just Docker.

## Kubernetes Basics

The basic idea of Kubernetes is to further abstract machines, storage, and networks away from their physical implementation. So it is a single interface to deploy containers to all kinds of clouds, virtual machines, and physical machines.

Here are a few of Kubernetes concepts to help understand what it does.

## Node

A node is a physical or virtual machine. It is not created by Kubernetes. You create those with a cloud operating system, like OpenStack or Amazon EC2, or manually install them. So you need to lay down your basic infrastructure before you use Kubernetes to deploy your apps. But from that point it can define virtual networks, storage, etc. For example, you could use OpenStack Neutron or Romana to define networks and push those out from Kubernetes.

## Pods

A pod is a one or more containers that logically go together. Pods run on nodes. Pods run together as a logical unit. So they have the same shared content. They all share the share IP address but can reach other other via localhost. And they can share storage. But they do not need to all run on the same machine as containers can span more than one machine. One node can run multiple pods.

Pods are cloud-aware. For example you could spin up two Nginx instances and assign them a public IP address on the Google Compute Engine (GCE). To do that you would start the Kubernetes cluster, configure the connection to GCE, and then type something like:

kubectl expose deployment my-nginx –port=80 –type=LoadBalancer

## Deployment

A set of pods is a deployment. A deployment ensures that a sufficient number of pods are running at one time to service the app and shuts down those pods that are not needed. It can do this by looking at, for example, CPU utilization.

## Vendor Agnostic

Kubernetes works with many cloud and server products. And the list is always growing as so many companies are contributing to the open source project. Even though it was invented by Google, Google is not said to dominate it’s development.

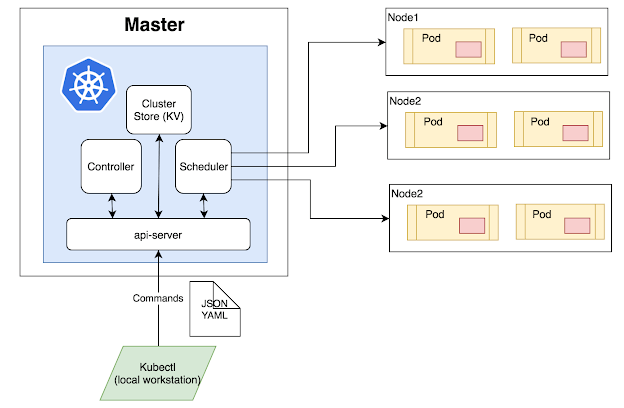
To illustrate, the OpenStack process to create block storage is called Cinder. OpenStack orchestration is called Heat. You can use Heat with Kubernetes to manage storage with Cinder.

Kubernetes works with Amazon EC2, Azure Container Service, Rackspace, GCE, IBM Software, and other clouds. And it works with bare-metal (using something like CoreOS), Docker, and vSphere. And it works with libvirt and KVM, which are Linux machines turned into hypervisors (i.e, a platform to run virtual machines).

## Use Cases

So why would you use Kubernetes on, for example, Amazon EC2, when it has its own tool for orchestration (CloudFormation)? Because with Kubernetes you can use the same orchestration tool and command-line interfaces for all your different systems. Amazon CloudFormation only works with EC2. So with Kubernetes you could push containers to the Amazon cloud, your in-house virtual and physical machines as well, and other clouds.

# Kubernetes Architecture



* Package the application and wrap the app into a container to give to cluster.
* Wrap Containerize app in a Pod because Kubernetes need containers wrapped as Pod.
* Pod Needs to wrap inside deployment for scaling and self healing.
* All instruction defines at YAML file, which is basically a way to describe the way app should work in Kubernetes.

## Master Node

It also known as head nodes and consider as brain or intelligence of the cluster. App should not deploy to Kubernetes to production without a highly available multi master control plane. Master node’s are responsible of running cluster. More than five master causes performance issue. Linux machine needs to run master node but can be anywhere. Every Master Node run’s a bunch of smaller service’s and each responsible for a single control plane feature and it is Microservice. It is consider as a good practice not to run user or Business Application on Master Node.

## Worker/Slave nodes

Worker nodes are another essential component which contains all the required services to manage the networking between the containers, communicate with the master node, which allows to assign resources to the scheduled containers.

It has three main components.

1. Kubelet

2. Container Runtime

3. Kube Proxy

# Virtual Kubelet (Nodeless Kubernetes)

Some cloud service provider provides Nodeless Kubernetes. A lot of Cloud platform have these days hosted container platform. Need to provide Containerized workload and cloud runs it. All you have to Pay what you run.

Post API Configuration in standard Kubernetes YAML file to API server on cloud and cloud run runs the kubelete.

Now we are going to setup a kubernetes Environment in our Local computer. Installing a kubernetes in local machine has some restriction, such as some features works in server machine or cloud based system. We are also going to use Docker containerization and a sample .net core Application for demonstration purpose.

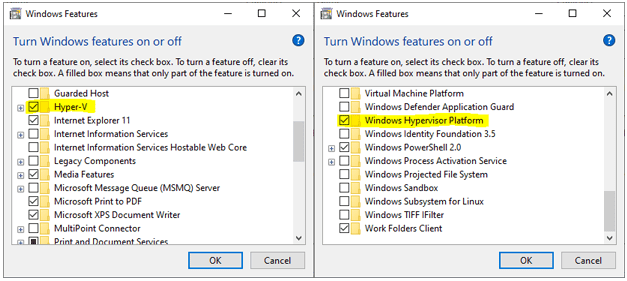
# Step 1: Install Kubectl.

1. Install Kubectl. - <https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/>
2. Check if kubectl installed: **kubectl version --short**

Note: kubectl has got a hidden directory called kube inside profile and it’s a yaml file.

# Step 2: Install & Setup Hyper-V

1. Go to Control Panel
2. On your left panel, click on **Programs**
3. Then click **Programs and Features** followed by **Turn Windows features on and off**.
4. Check **Hyper-V** and **Windows Hypervisor Platform**
5. Click **OK**



System will now start installing Hyper-V on the background, it may need to reboot a couple of times until everything is configured properly.

Run the following command as Administrator on powershell and verify if Hyper-V is installed successfully on your machine:

Get-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V

# Step 3: Install Docker for Windows

To install Docker just go to <https://docs.docker.com/desktop/windows/install/>

**Install**

Double-click Docker for Windows Installer to run the installer.

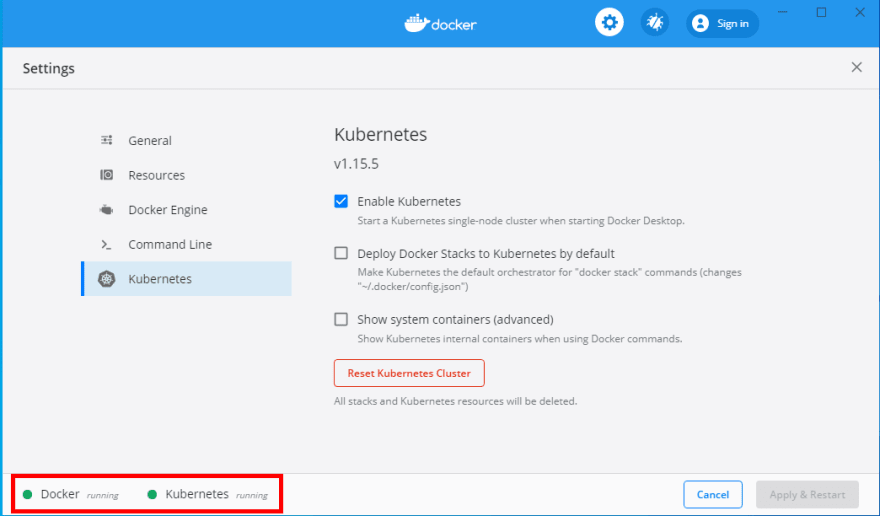
When the installation finishes, Docker starts automatically. The whale in the notification area indicates that Docker is running, and accessible from a terminal.

# Step 4: Install Kubernetes on Windows 10

Now follow the instructions to install Kubernetes.

1. Right-click the Docker tray icon
2. Click "Settings"
3. On the left panel click "Kubernetes"
4. Check Enable Kubernetes and click "Apply"

It may take around 5~10 minutes and the installation time depends on your Internet speed and your PC performance.



Some commands you can check –

* docker info
* kubectl cluster-info
* docker container ls
* kubectl get nodes //shows list of nodes

# Step 5: Aim of this document

We are going to deploy a .Net Core web API and React App inside Pods. These sample apps were built with graphql. We are going to deploy our api inside a pod and will use the route to fetch data in frontend application.

We are not going to build the API and Frontend app in this documentation. If you want to learn the process to build .net core web API and Frontend APP with graphql, please check following documentations of RND-1 in ASA git repository.

1. Net Core 5 Web Api with GraphQL(Step by Step)
2. Configuring ReactJS with GraphQL for Frontend Application
3. Configuring ReactJS with GraphQL using Apollo Client for Frontend Application *(you can also read this document, if you want to build your frontend app with graphQL using Apollo Client)*

**\*\*\*We are going to use Windows PowerShell to write all Docker and Kubectl commands\*\*\***

# Step 6: Clone or Download the Repository for API

Clone or download the repo - <http://192.168.100.40:8080/Repository/fe8dd758-727b-499b-a9f5-fe069065eff4/master/Tree/src/Graphql/NetCore_GraphQL>

open the project and restore nuget packages. Or create a new project and follow according to **Net Core 5 Web Api with GraphQL(Step by Step)** document.

# Step 7: Clone or Download the Repository for Frontend Application

Clone or download the repo - <http://192.168.100.40:8080/Repository/fe8dd758-727b-499b-a9f5-fe069065eff4/master/Tree/src/Graphql/react_graphql>

open the project and restore nuget packages. Or create a new project and follow according to **Configuring ReactJS with GraphQL for Frontend Application** document.

# Step 8: Configuring Local Registry

One of the vital part of this documentation is to create our own Local Registry to store Images instead of Docker hub. We don’t want to share our Application images on any website such as Docker Hub, because of that purpose we are going to create our own Registry.

Local Registry is nothing but an image and container. Please follow below steps to create your own registry.

1. If you are using Windows OS, please open Windows Powershell and write below command to create a Local Registry.

**docker run -d -p 5000:5000 --restart=always --name registry registry:2.7**

1. We are reserving port 5000 for local registry port and name the image and container **registry**.
2. We can check Registry logs with below command;

**docker logs -f registry**

1. Later we will push and pull images inside registry.

# Step 9: Configuring .Net Core Web API for Kubernetes Cluster

Please follow following steps to configure .Net Core Web API with Docker and Kubernetes.

## 9.1: Setup a Dockerfile.

Create a file without any file-extension and name the file **Dockerfile** and write below code.

FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base

WORKDIR /app

EXPOSE 5001

ENV ASPNETCORE\_URLS=http://\*:5001

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build

WORKDIR /src

COPY ["NetCore\_GraphQL.csproj", "./"]

RUN dotnet restore "./NetCore\_GraphQL.csproj"

COPY . .

WORKDIR "/src/."

RUN dotnet build "NetCore\_GraphQL.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "NetCore\_GraphQL.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "NetCore\_GraphQL.dll"]

You can also create a **.dockerignore** file in your root directory and write below code.

\*\*/.classpath

\*\*/.dockerignore

\*\*/.env

\*\*/.git

\*\*/.gitignore

\*\*/.project

\*\*/.settings

\*\*/.toolstarget

\*\*/.vs

\*\*/.vscode

\*\*/\*.\*proj.user

\*\*/\*.dbmdl

\*\*/\*.jfm

\*\*/azds.yaml

\*\*/bin

\*\*/charts

\*\*/docker-compose\*

\*\*/Dockerfile\*

\*\*/node\_modules

\*\*/npm-debug.log

\*\*/obj

\*\*/secrets.dev.yaml

\*\*/values.dev.yaml

LICENSE

README.md

## 9.2 Build and Run Docker Image

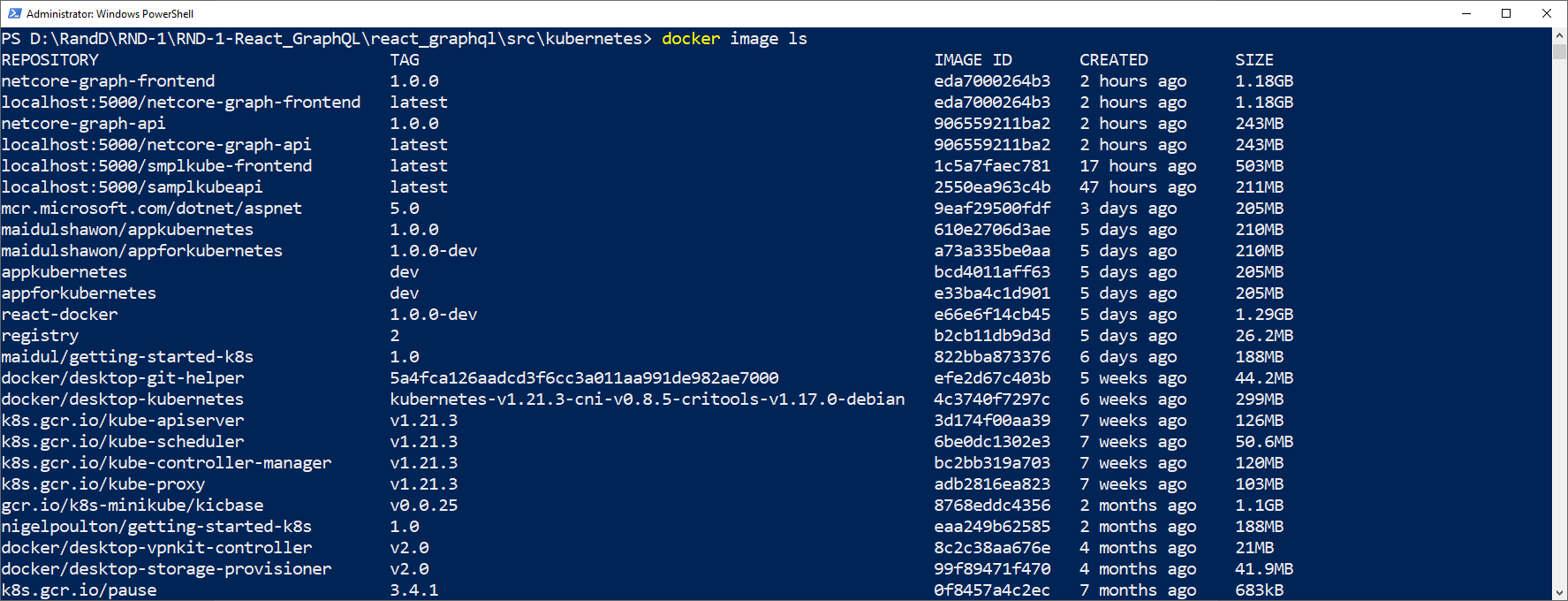
As we have created Dockerfile, now we have to build our App into a Docker Image.

Build Docker image command: **docker build -t netcore-graph-api:1.0.0 .**

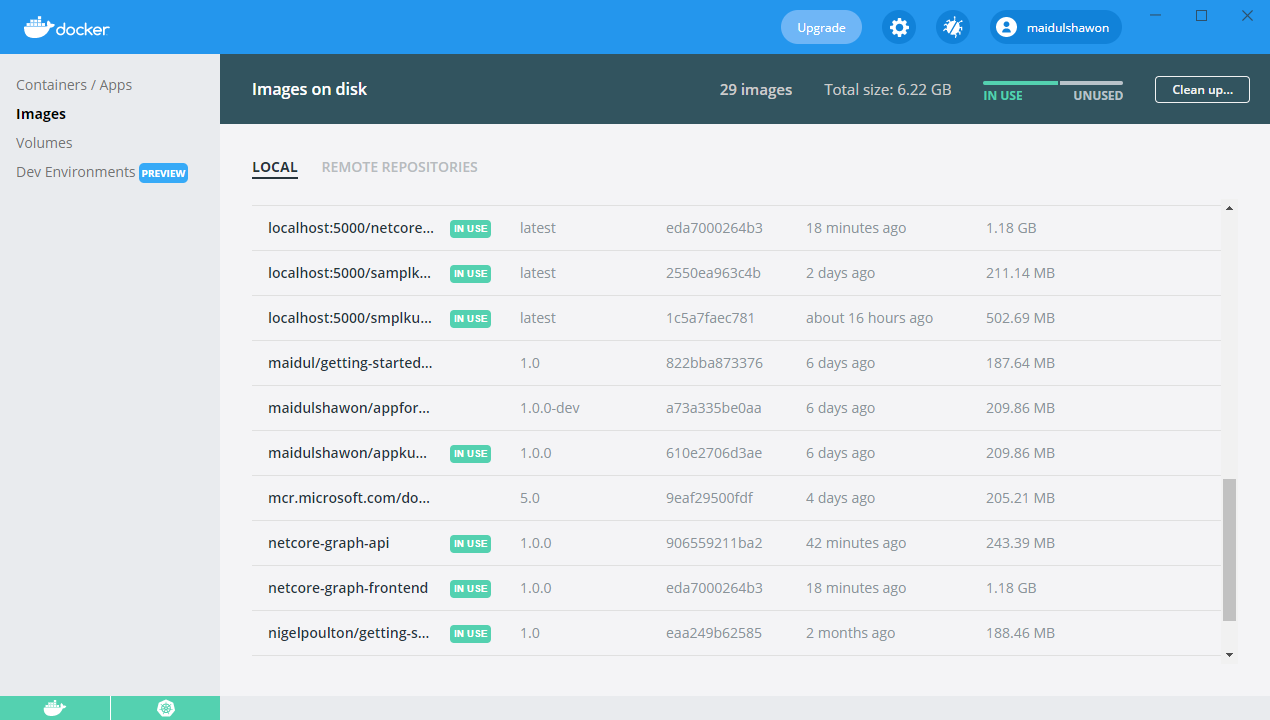
Now is the time to check, if the image created properly. Use below command to check all Docker images in the computer.

**docker image ls** or **docker images**

above command will display list of images with our created image like below.



You can also check your image in your Docker Desktop App. Open the Docker Desktop App and go to Images Tab.

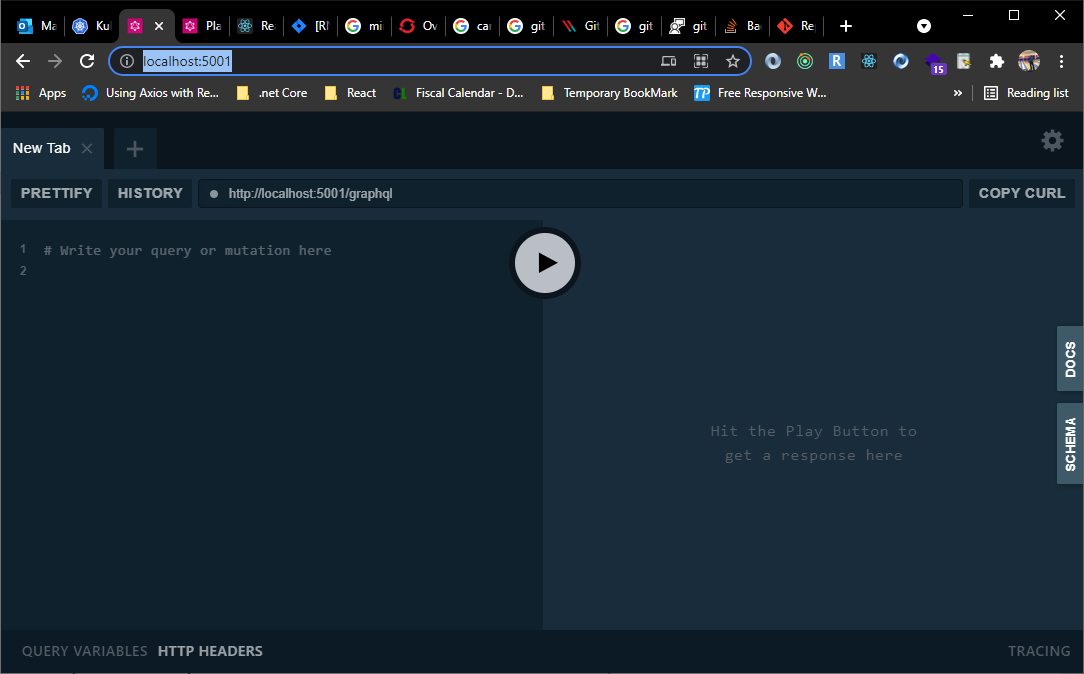


If you want to run your app inside a container, please use below command.

**docker run -dp 5001:5001 --name netcore-graph-api-temp netcore-graph-api:1.0.0**

Above command will create a container and will run the image inside container. As we have used port **5001**, you can check on browser with URL **http://localhost:5001/**

Browser will show a display like below if you are using same API we are using for this documentation.



## 9.3 Push to or pull from our Local Registry

We have created our local registry earlier, now is the time to push an image inside local registry.

1. To push to or pull from our local registry, we need to add the registry’s location to the repository name. The format is as follows: my.registry.address:port/repositoryname
2. In our example, we need to replace my.registry.address:port with **localhost:5000** because our registry is running on our localhost and is listening on **port 5000**. Here is the full repository name: **localhost:5000/netcore-graph-api**. We have to run below Docker Tag command:

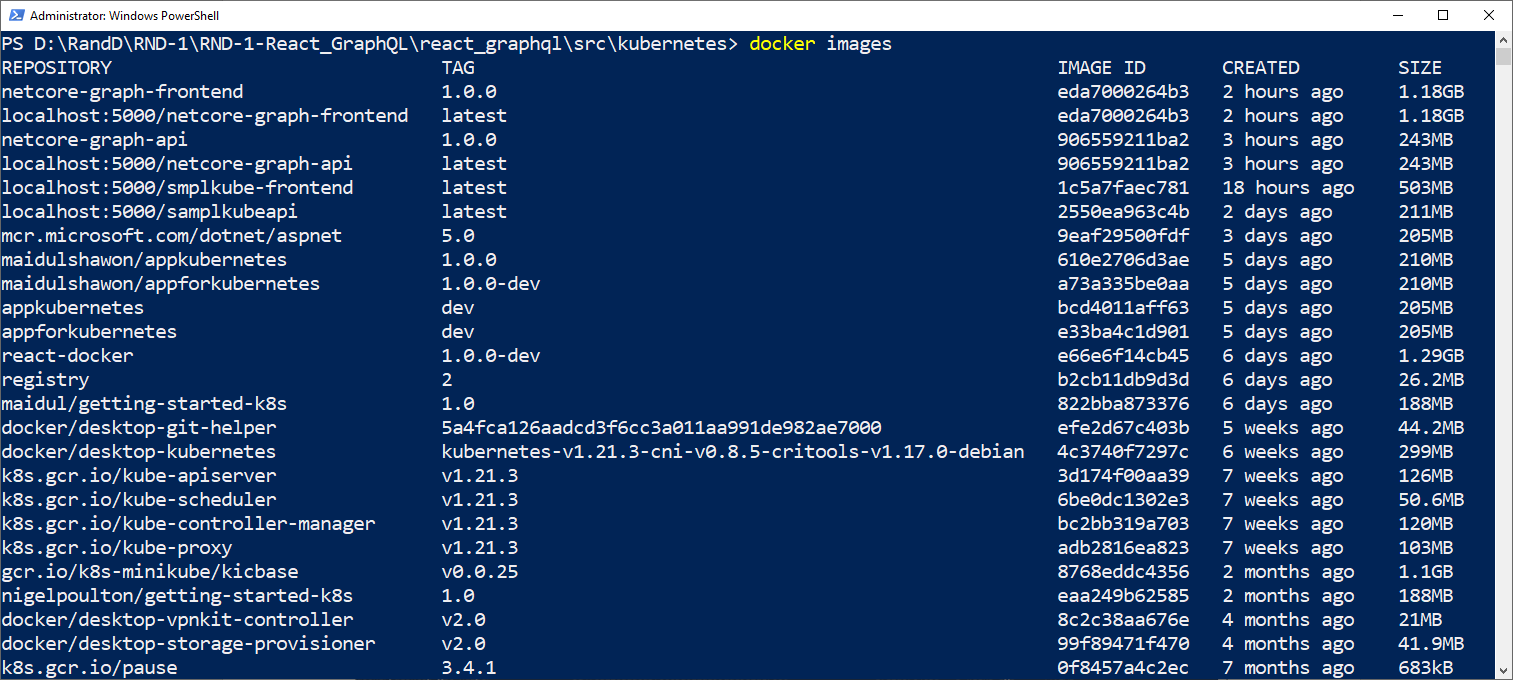
**docker tag netcore-graph-api:1.0.0 localhost:5000/netcore-graph-api**

1. Use below command to push to our local Registry.

**docker push localhost:5000/netcore-graph-api**

1. We can check our newly pushed repository with below command and will show a display like below.

**docker image ls** or **docker images**



As we can see that the repository has created with default **latest** tag.

1. As our image repository has been created, we can pull from Local Registry whenever we needed with below command.

**docker pull localhost:5000/netcore-graph-api**

## 9.4 Creating a Deployment

We have already created image repository, now is the time to create a yml file to describe to the kubernetes cluster about pods, containers with repository image. The Yaml file contains the description of every Kubernetes object (service, pods, deployment, etc.) The Kubernetes entity makes sure our application will have as many replicas (parallel pods) as we define. We can also define the type of Docker image we want to use, what type of ports are used, and metadata for our application

1. Create a Folder inside root directory of your .net core API project and name the folder **kubernetes**
2. Create another Folder inside Kubernetes folder and name it **deployments**
3. Create a YML file and name the file **deployment.yml.**
4. Write below code inside **deployment.yml** file. ***(code is commented with detail)***

apiVersion: apps/v1

kind: Deployment

metadata:

#Name of the deployment

  name: netcore-graph-api-deployment

spec:

#Number of pods

  replicas: 2

  selector:

    matchLabels:

    #Name of pods

      app: netcore-graph-api-pod

  template:

    metadata:

      labels:

        app: netcore-graph-api-pod

    spec:

      containers:

      #Name of container

      - name: netcore-graph-api-container

      #pulling image from Local Registry

        image: localhost:5000/netcore-graph-api

        resources:

        #resource configuration

          limits:

            memory: "128Mi"

            cpu: "500m"

        ports:

        #Port exposed from dockerfile or from api

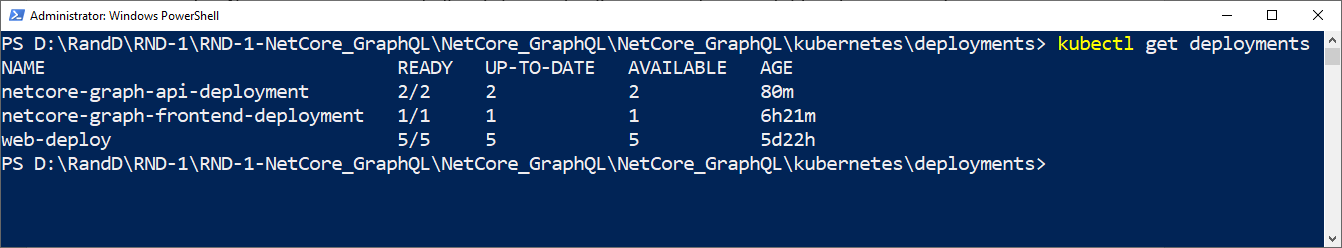
        - containerPort: 5001

1. Now open your powershell and change the directory to the same folder where you created **deployment.yml** file. In our case, we have to change the directory to deployments folder.
2. Write below command to proceed the **deployment.yml** file.

**kubectl apply -f deployment.yml**

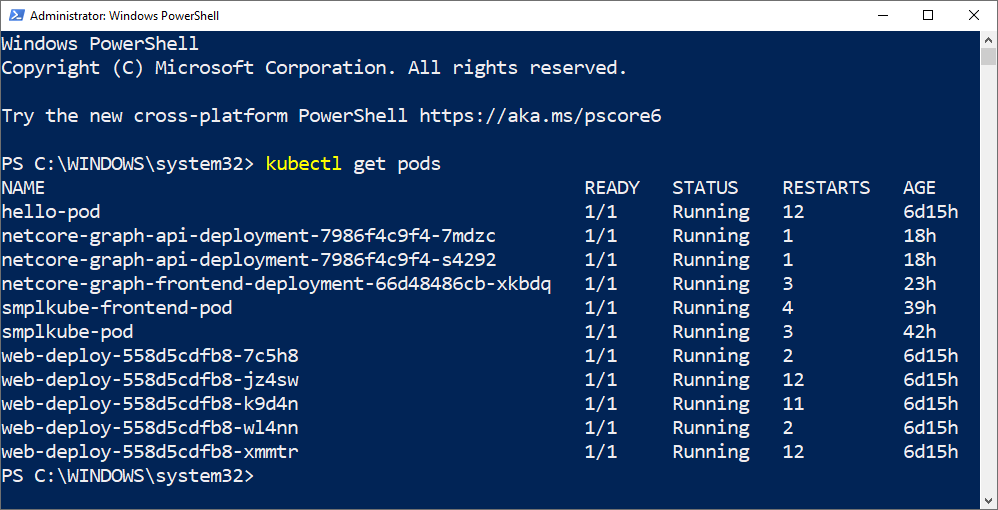
1. Now you can check all running deployments with below command.

**kubectl get deployments**



1. Command to check all running pods.

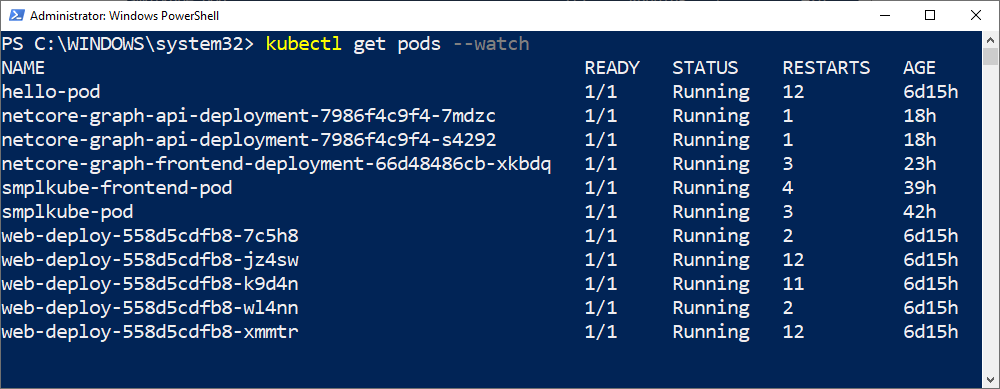
**kubectl get pods**



Multiple pods are running for netcore-graph-api-deployment because we have mentioned in *yml* file to use *2 replicas* in our deployment file.

1. Command to monitor pods. Below command will show the continuous update of pods.

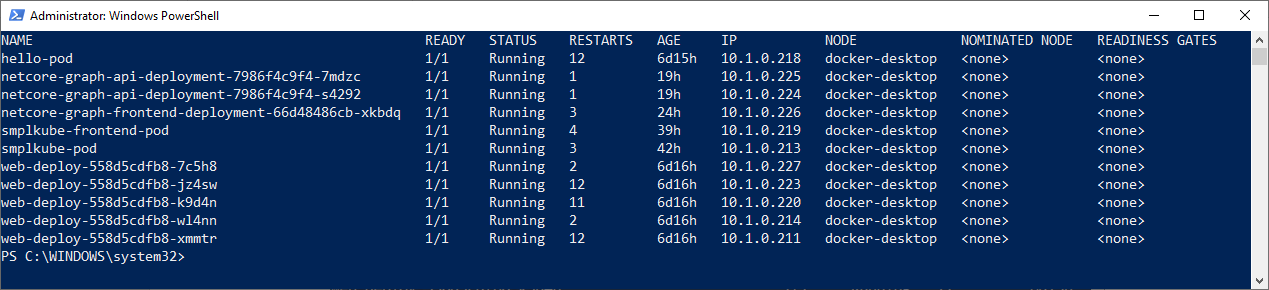
**kubectl get pods –watch**



Use (*CTRL+C)* to stop monitoring of pods.

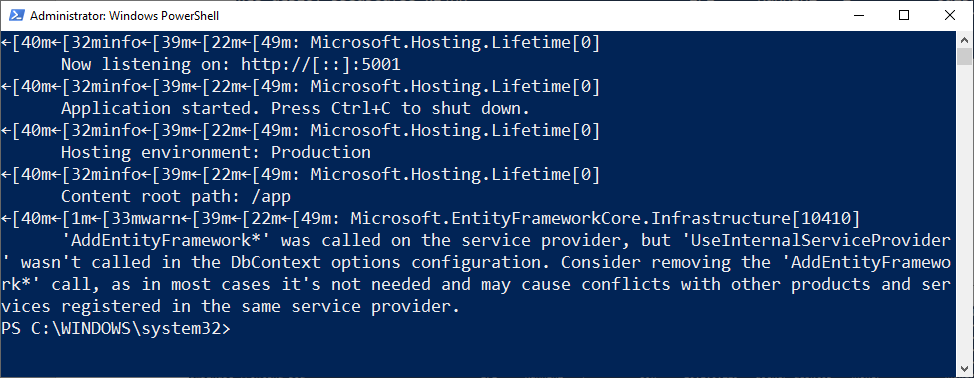
1. Command to check pod status with pods IP address.

**kubectl get pods -o wide**



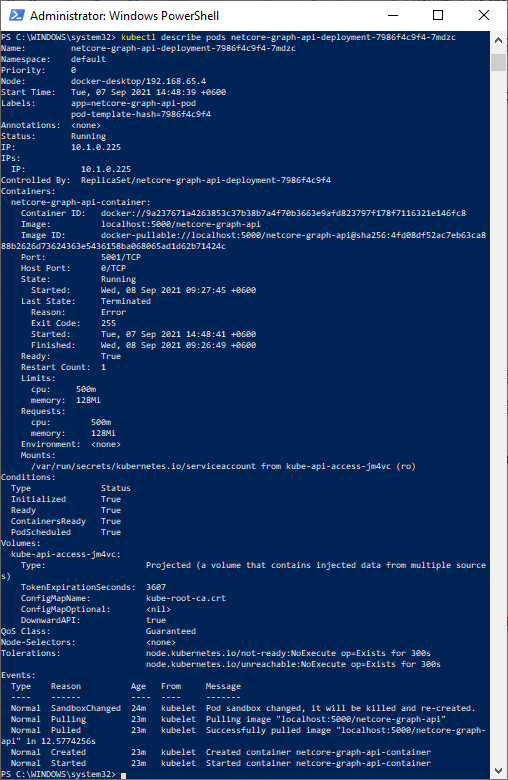
1. Command to check pods Log

**kubectl logs netcore-graph-api-deployment-7986f4c9f4-7mdzc**



1. Complete details of pod can be check by below command.

**kubectl describe netcore-graph-api-deployment-7986f4c9f4-7mdzc**



## 9.5 Defining a Service

1. Create a Folder inside **Kubernetes** folder and name it **services.**
2. Create a YML file and name the file **service.yml.**
3. Write below code inside **service.yml** file. ***(code is commented with detail)***

apiVersion: v1

kind: Service

metadata:

    #Name of the service

  name: netcore-graph-api-service

spec:

  selector:

  #select Kubernetes resources based on the value of labels and resource fields

  #assigned to a group of pods or nodes.

  #here we are writting selector>app same as selector>app from deployment.yml

    app: netcore-graph-api-pod

  ports:

  #exposes the Kubernetes service on the specified port within the cluster

  - port: 5005

  #This is the actual port on which our application is running

  #inside the container.

    targetPort: 5001

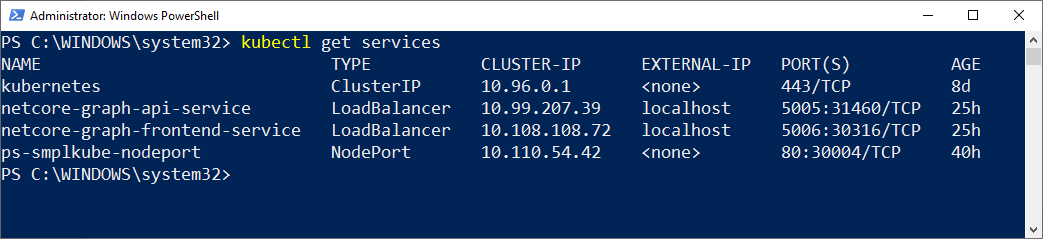
  type: LoadBalancer

1. Now open your powershell and change the directory to the same folder where you created **service.yml** file. In our case, we have to change the directory to services folder.
2. Write below command to proceed the **service.yml** file.

**kubectl apply -f service.yml**

1. Now you can check all running services with below command.

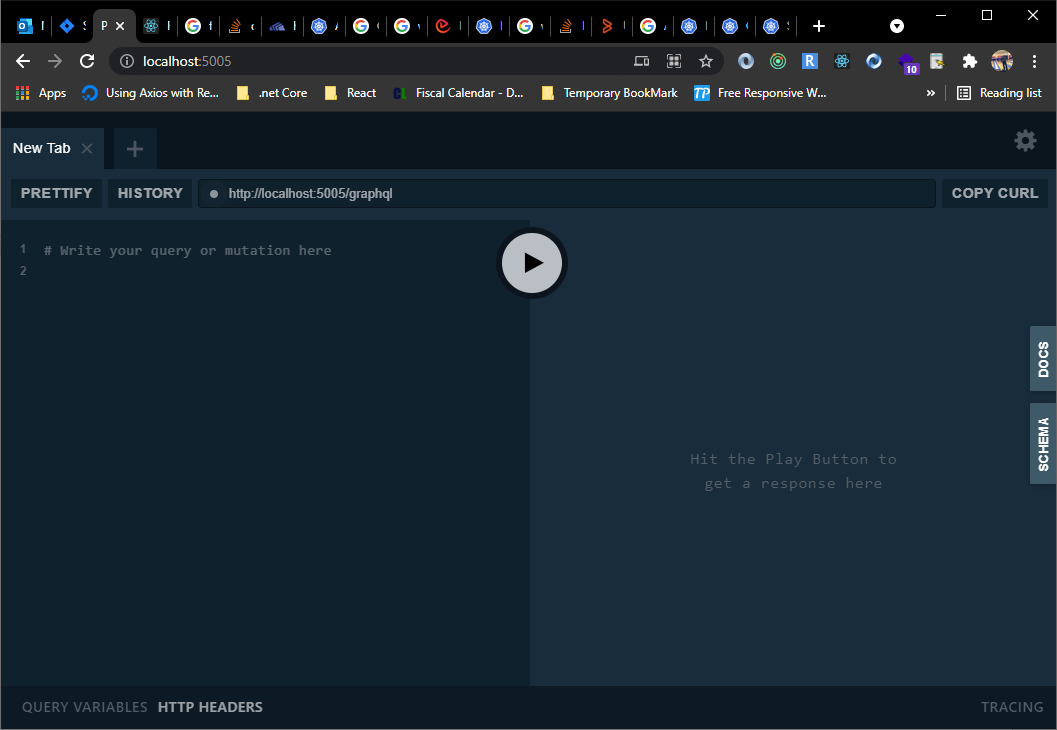
**kubectl get services**



## 9.6 Checking on Browser

In above list, we can see a column for port(s) number. We can now browse our application with localhost:**PORT\_NUMBER.** As we can see from the list of services that the port number of our created service is **5006**.

1. Open your browser and visit - <http://localhost:5005/>
2. You will see a browser like below.



Browser will show Net Core GraphQL API Playground. You can write graphql queries and test on the playground.

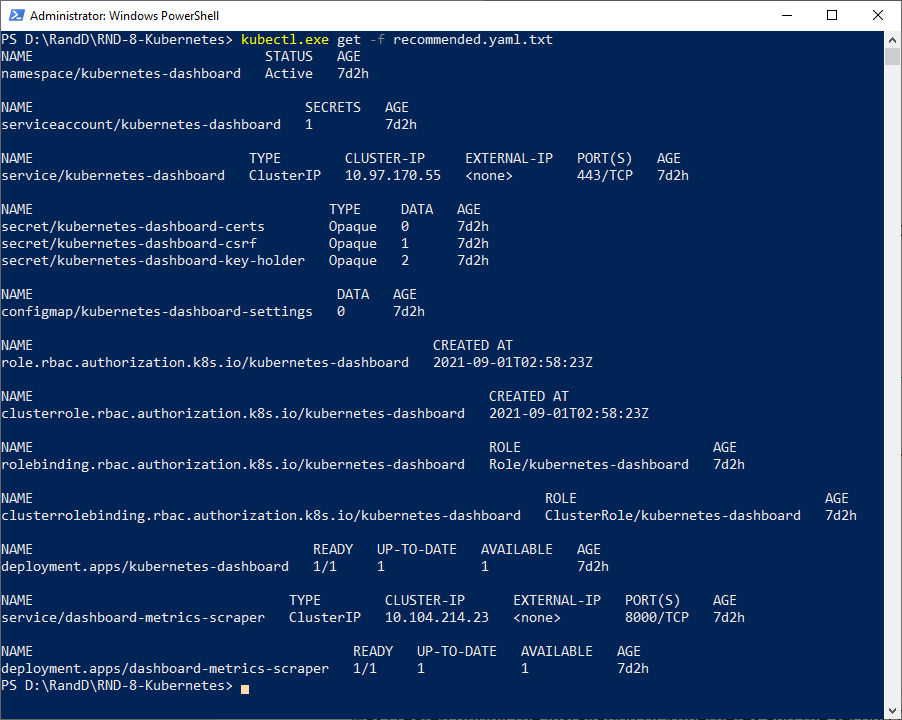
As we can see that our API is now running on Kubernetes cluster, now we are going to configure frontend application in a new pod.

Before we start deploying Frontend app inside the cluster we are going to configure Kubernetes Dashboard to see a GUI.

# Step 10: Install Kubernetes Dashboard

Run the following commands to deploy & enable the Kubernetes Dashboard using the following commands.

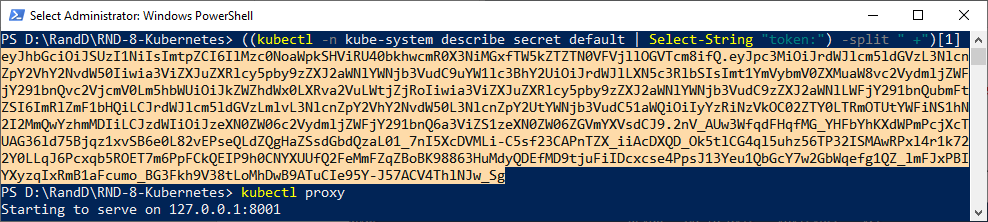
1. Download the YAML configuration file from <https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-rc7/aio/deploy/recommended.yaml>
2. Change the directory of powershell to the directory you saved above YAML file.
3. Write following command: kubectl apply -f recommended.yaml.txt
4. Run following command to verify: kubectl.exe get -f .\recommended.yaml.txt



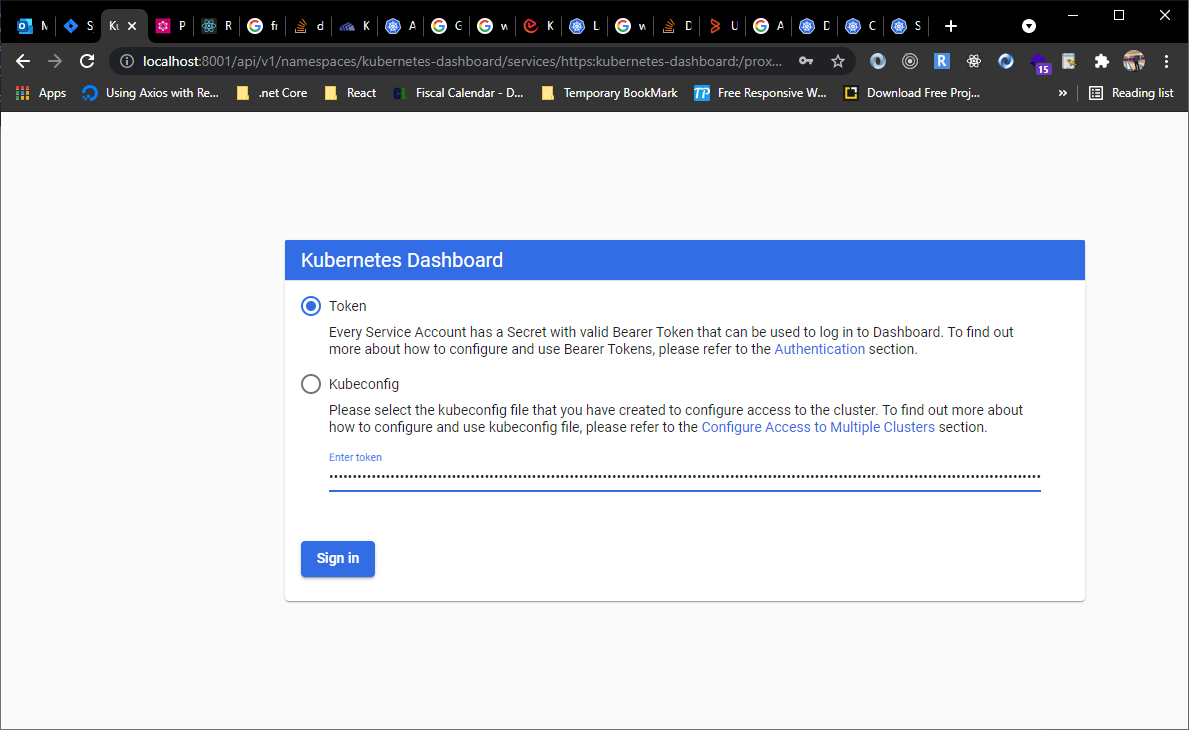
# Step 10: Access the dashboard

There are two ways to access the dashboard with tokens, the first one is using the default token that was created during the installation of Kubernetes and the second (more secure) way is by creating users, giving them permissions, and then get the generated token. In the name of simplicity, we will use the first way.

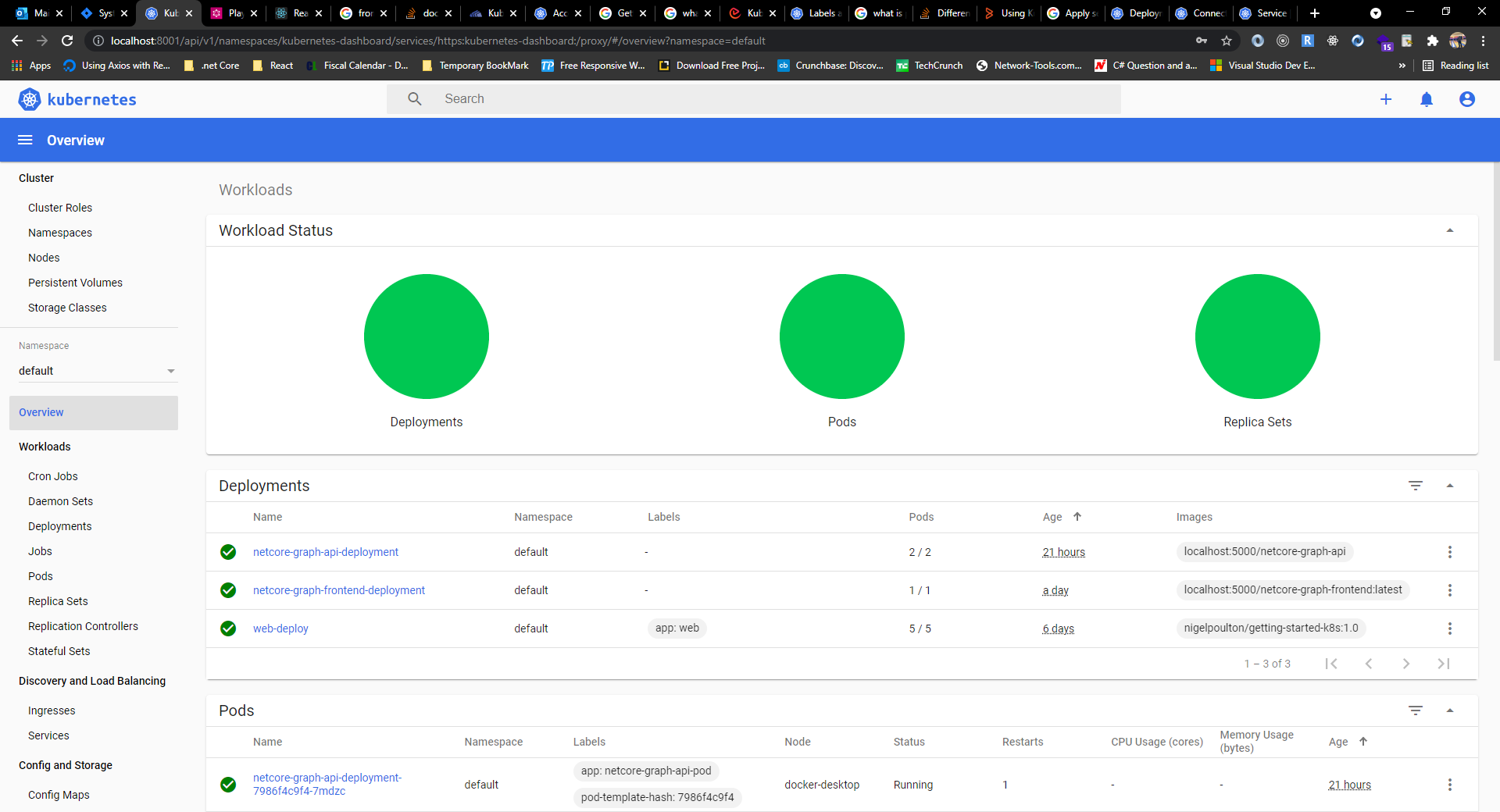
1. Run the following command powershell: ((kubectl -n kube-system describe secret default | Select-String "token:") -split " +")[1]
2. Above command will generate a token.
3. Copy the generated token
4. Run kubectl proxy.



1. Open the following link on your browser: <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>
2. Select Token & paste the generated token
3. Sign In



If everything configured properly, you'll be able to see a dashboard and your cloud resources.

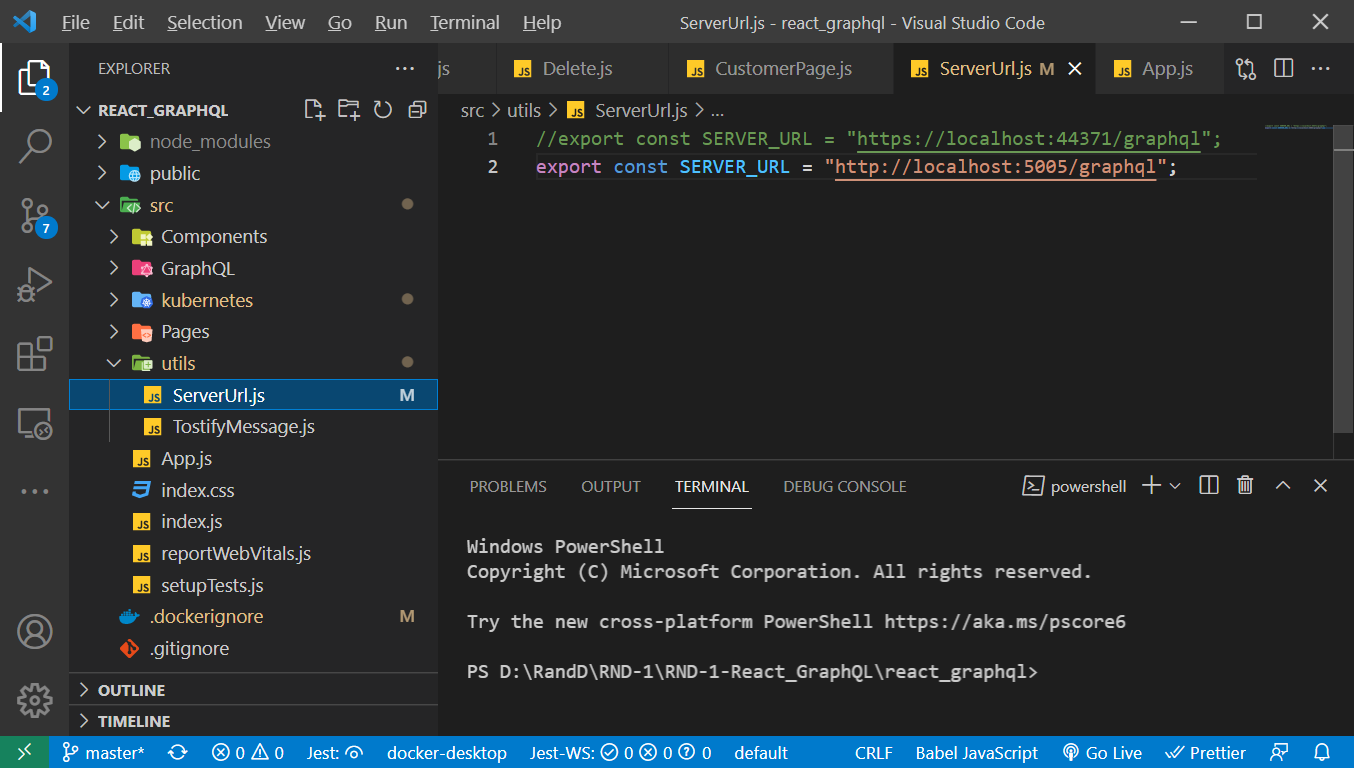


Now you browse the GUI of kubernetes. you will find there list of deployments, pods and replica sets. You can click cluster tab on left pane and will find list of cluster roles, Namespaces, Nodes and Storage Classes. You will find one node –(docker-desktop) in the list of Nodes.

# Step 11: Configuring ReactJS Frontend APP for Kubernetes Cluster

**Before we start configuring dockerfile and kubenetes cluster, please open Frontend Application you cloned - *RND-1-React\_GraphQL.***

1. Go to **src>utils** folder.
2. Open **ServerUrl.js** file and change the server URL same as hosted API port.
3. As we are using GrphQL for API, our SERVER\_URL will be **http://localhost:5005/graphql**



Please follow following steps to configure ReactJS Frontend APP with Docker and Kubernetes.

## 11.1: Setup a Dockerfile

Create a file in the root directory without any file-extension and name the file **Dockerfile** and write below code.

# pull official base image

FROM node:latest

# A directory within the virtualized Docker environment

# Becomes more relevant when using Docker Compose later

WORKDIR /usr/src/app

# Copies package.json and package-lock.json to Docker environment

COPY package\*.json ./

# Installs all node packages

RUN npm install

# Copies everything over to Docker environment

COPY . .

# Finally runs the application

CMD [ "npm", "start" ]

You can also create a **.dockerignore** file in your root directory and write below code.

/node\_modules

/build

.git

\*.md

.gitignore

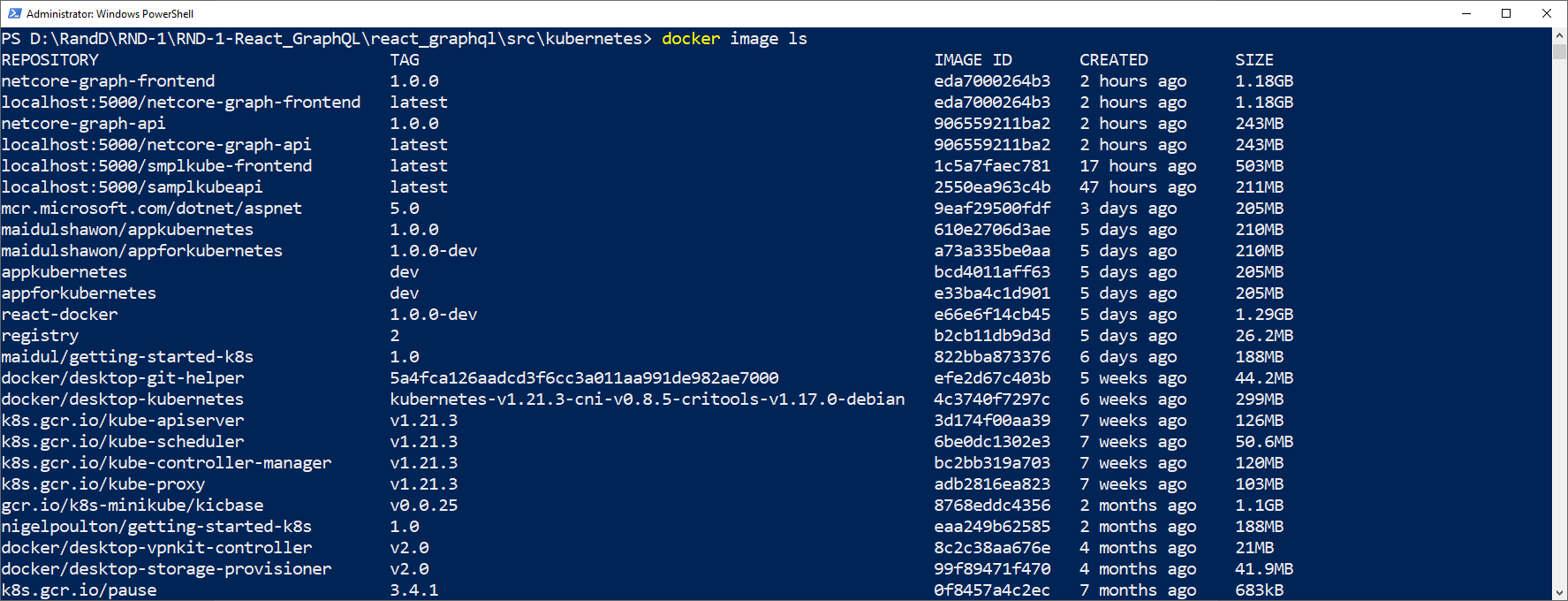
## 11.2 Build and Run Docker Image

As we have created Dockerfile, now we have to build our App into a Docker Image.

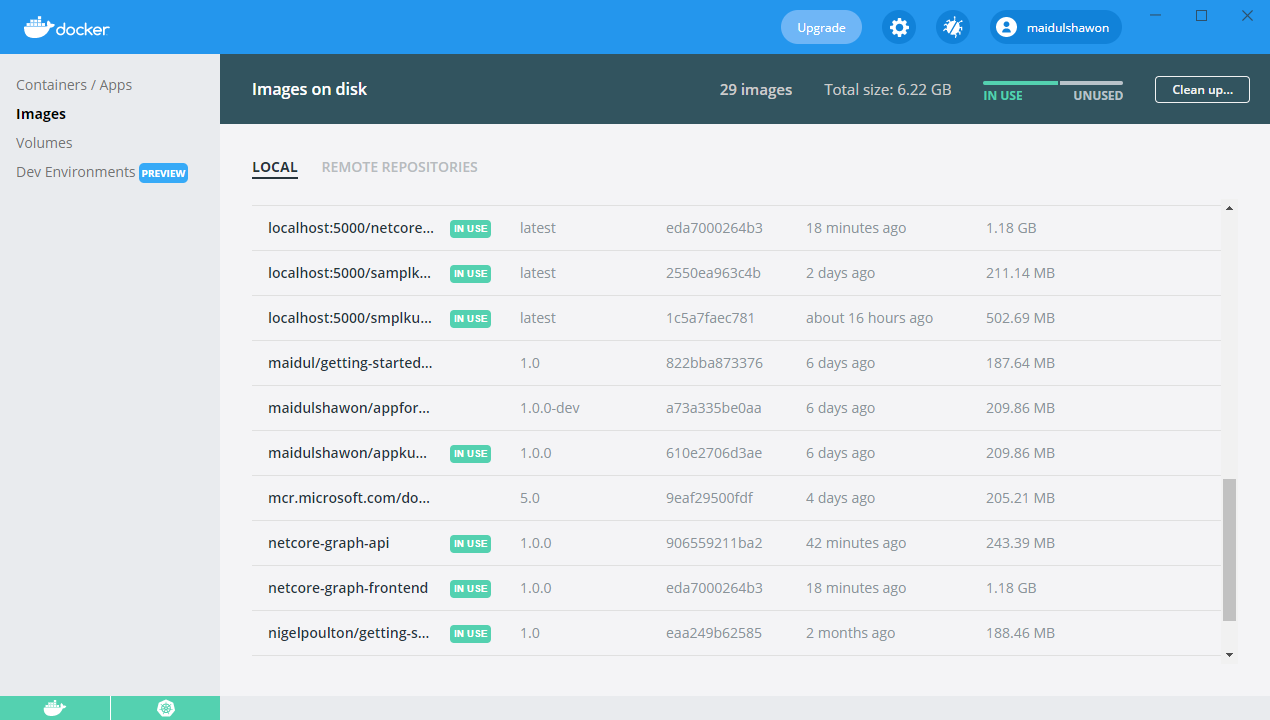
1. Build Docker image command: **docker build -t netcore-graph-frontend:1.0.0 .**
2. Now is the time to check, if the image created properly. Use below command to check all Docker images in the computer.

**docker image ls** or **docker images**

above command will display list of images with our created image like below.



You can also check your image in your Docker Desktop App. Open the Docker Desktop App and go to Images Tab.

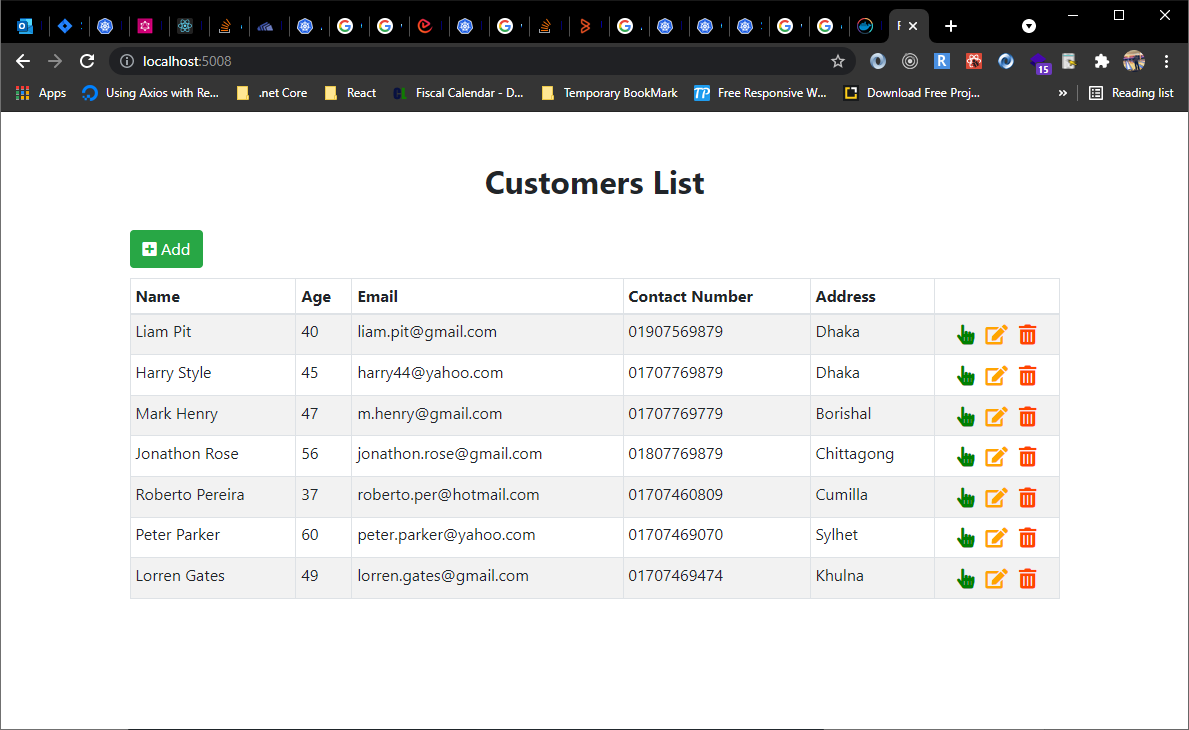


If you want to run your app inside a container, please use below command.

**docker run -dp 5008:3000 --name netcore-graph-frontend netcore-graph-frontend:1.0.0**

Above command will create a container and will run the image inside container. As we have used port **5008**, you can check on browser with URL **http://localhost:5008/**

Browser will show a display like below if you are using same Frontend Application and .Net Core API we are using for this documentation.



## 11.3 Push to or pull from our Local Registry

1. To push to or pull from our local registry, we need to add the registry’s location to the repository name. The format is as follows: my.registry.address:port/repositoryname
2. In our example, we need to replace my.registry.address:port with **localhost:5000** because our registry is running on our localhost and is listening on **port 5000**. Here is the full repository name: **localhost:5000/netcore-graph-frontend**. We have to run below Docker Tag command:

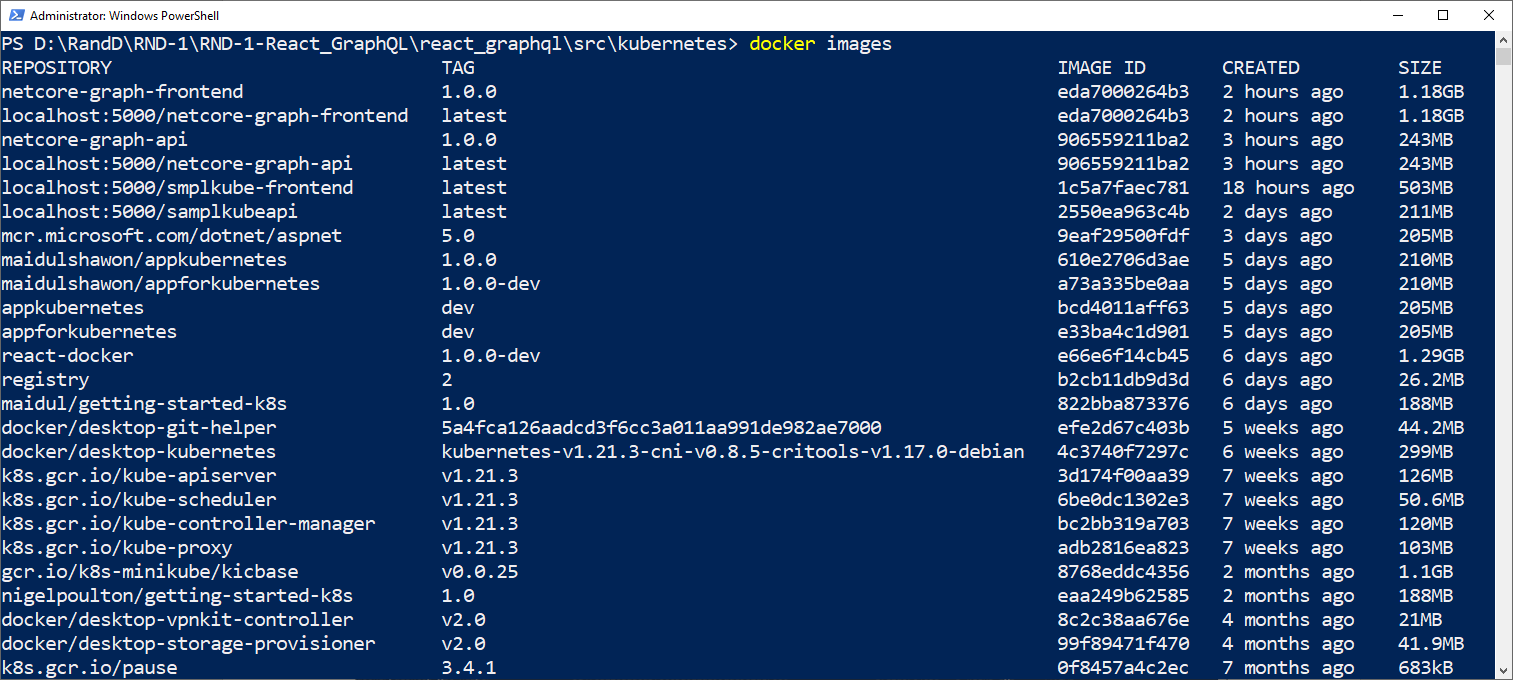
**docker tag netcore-graph-frontend:1.0.0 localhost:5000/netcore-graph-frontend**

1. Use below command to push to our local Registry.

**docker push localhost:5000/netcore-graph-frontend**

1. We can check our newly pushed repository with below command and will show a display like below.

**docker image ls** or **docker images**



As we can see that the repository has created with default **latest** tag.

1. As our image repository has been created, we can pull from Local Registry whenever we needed with below command.

**docker pull localhost:5000/ netcore-graph-frontend**

## 11.4 Creating a Deployment

We have already created image repository, now is the time to create a **YML** file to describe to the kubernetes cluster about pods, containers with repository image.

1. Create a Folder inside **src** directory of your React\_GraphQL project and name the folder **kubernetes**
2. Create a YML file and name the file **deployment.yml.**
3. Write below code inside **deployment.yml** file. ***(code is commented with detail)***

apiVersion: apps/v1

# Name of the deployment

kind: Deployment

metadata:

# Name of pods

  name: netcore-graph-frontend-deployment

spec:

  selector:

    matchLabels:

      app: netcore-graph-frontend-pod

  template:

    metadata:

      labels:

        app: netcore-graph-frontend-pod

    spec:

      containers:

      #Name of container

        - name: netcore-graph-frontend-container

        #pulling image from Local Registry

          image: localhost:5000/netcore-graph-frontend:latest

          ports:

          #Port exposed from dockerfile or from api

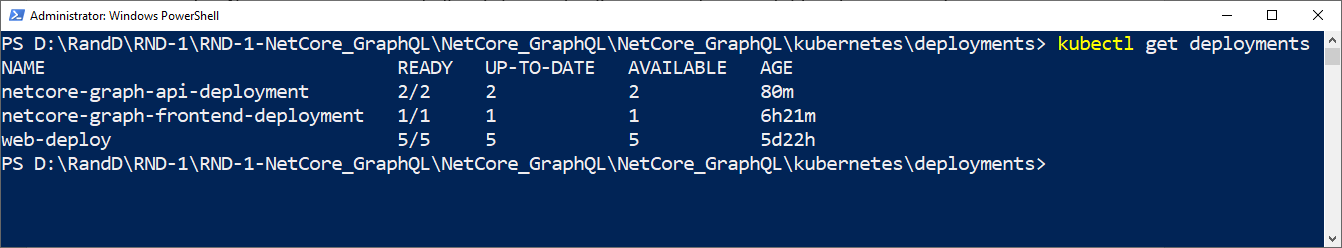
            - containerPort: 3000

1. Now open your powershell and change the directory to the same folder where you created **deployment.yml** file. In our case, we have to change the directory to deployments folder.
2. Write below command to proceed the **deployment.yml** file.

**kubectl apply -f deployment.yml**

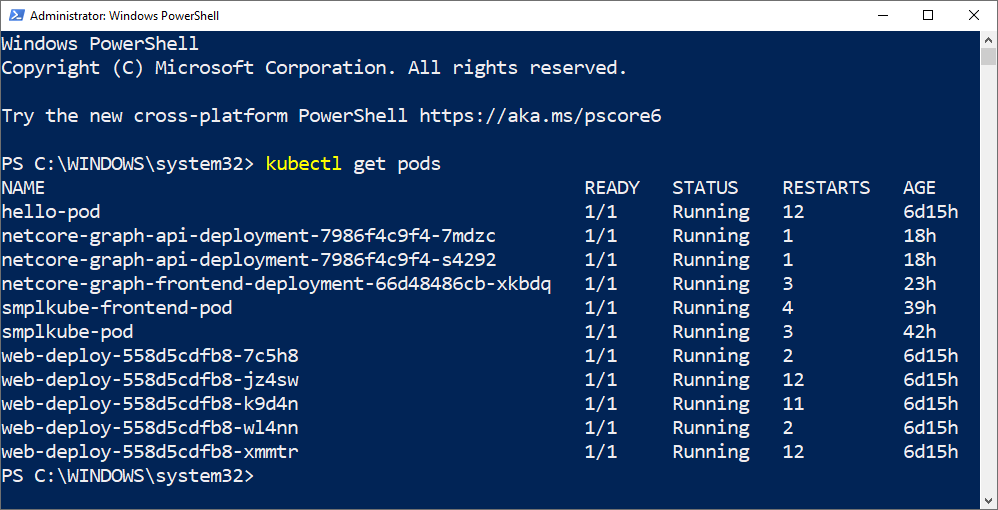
1. Now you can check all running deployments with below command.

**kubectl get deployments**



1. Command to check all running pods.

**kubectl get pods**



## Defining a Service

1. Create a YML file and name the file **services.yml**
2. Write below code inside **services.yml** file. ***(code is commented with detail)***

apiVersion: v1

kind: Service

metadata:

  # Name of the service

  name: netcore-graph-frontend-service

spec:

  selector:

    #select Kubernetes resources based on the value of labels and resource fields assigned to a group of pods or nodes.

    #here we are writting selector>app same as selector>app from deployment.yml

    app: netcore-graph-frontend-pod

  ports:

    #exposes the Kubernetes service on the specified port within the cluster

    - port: 5006

      #This is the actual port on which our application is running inside the container.

      targetPort: 3000

      # Exposes the Service externally

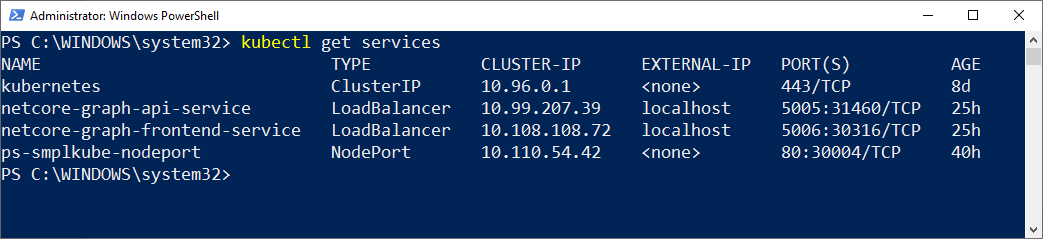
  type: LoadBalancer

1. Now open your powershell and change the directory to the same folder where you created **service.yml** file. In our case, we have to change the directory to services folder.
2. Write below command to proceed the **service.yml** file.

**kubectl apply -f service.yml**

1. Now you can check all running services with below command.

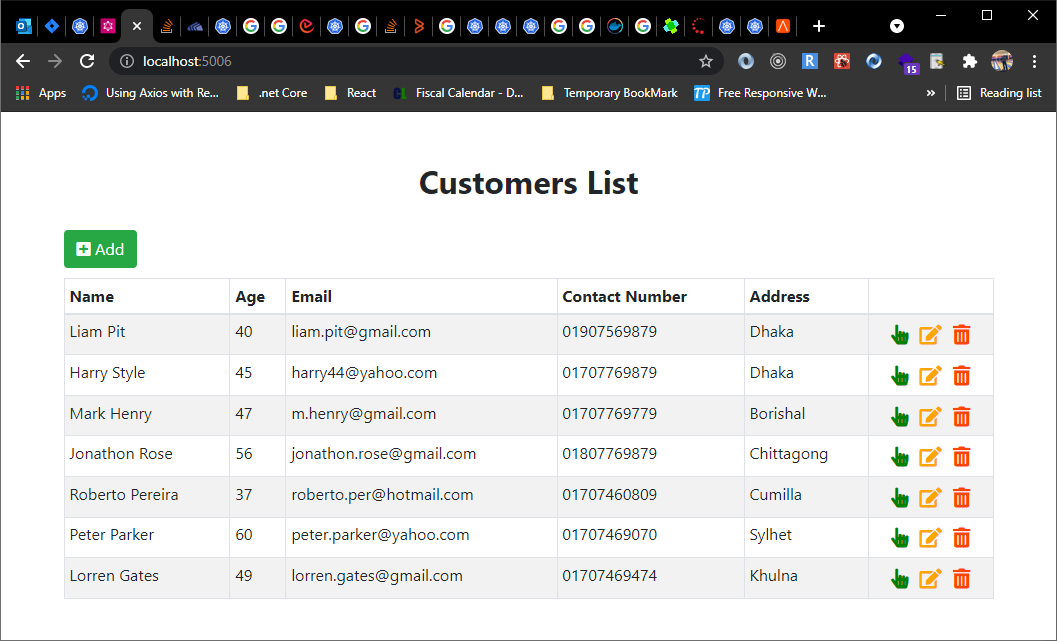
**kubectl get services**



## 11.6 Checking on Browser

In above list, we can see a column for port(s) number. We can now browse our application with localhost:**PORT\_NUMBER.** As we can see from the list of services that the port number of our created service is **5006**.

1. Open your browser and visit - <http://localhost:5006/>
2. You will see a browser like below.



Browser will show Customer List table and all these data is fetching from Net Core GraphQL API. You will be able to do CRUD operation on this Application.

# Some other Commands

**kubectl get rs - it shows desired and current pods.**

Exporting the Filesystem

docker export –o [file\_name\_you\_want\_to\_set].tar [containerID]

example command: **docker export -o frontendapp.tar cff281470546**

Stop a local registry

To stop the registry, use the same docker container stop command as with any other container.

**docker container stop registry**

To remove the container, use docker container rm.

**docker container stop registry && docker container rm -v registry**

Check files inside containers

docker run -t -i netcore-graph-frontend:1.0.0 /bin/bash

write **ls** command to find all folders

Prune images

The docker image prune command allows you to clean up unused images. By default, docker image prune only cleans up dangling images. A dangling image is one that is not tagged and is not referenced by any container. To remove dangling images:

docker image prune

Prune containers

When you stop a container, it is not automatically removed unless you started it with the --rm flag. To see all containers on the Docker host, including stopped containers, use docker ps -a. You may be surprised how many containers exist, especially on a development system! A stopped container’s writable layers still take up disk space. To clean this up, you can use the docker container prune command.

docker container prune

# References

* <https://docs.docker.com/get-started/overview/>
* <https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html>
* <https://kubernetes.io/docs/concepts/overview/components/>
* <https://www.guru99.com/kubernetes-tutorial.html>
* [https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-cluster#:~:text=A%20desired%20state%20is%20defined,defined%20with%20the%20Kubernetes%20API](https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-cluster).
* <https://yannalbou.medium.com/kubernetes-desired-state-4c5c4e873743>
* <https://downey.io/blog/desired-state-vs-actual-state-in-kubernetes/>
* <https://www.baeldung.com/ops/docker-container-filesystem>
* https://docs.docker.com/registry/deploying/