**Exercise - Push a microservice image to Docker Hub**

In order for Kubernetes to create a container image, it needs a place to get it from. Docker Hub is a central place to upload Docker images. Many products, including Kubernetes, can create containers based on images in Docker Hub.

**Important**

To complete this exercise, you will need the Docker Desktop application, a Docker Hub account, and a text editor such as [**Visual Studio Code**](https://code.visualstudio.com/).

* Download [**Docker Desktop here**](https://www.docker.com/products/docker-desktop).
* Follow the [**directions here to sign up for a Docker Hub account**](https://hub.docker.com/).

**Retrieve the Contoso Pizza Shop microservice container images**

The code for the Contoso Pizza Shop and the Dockerfiles to build the container images has already been created for you. Clone the [repository from GitHub](https://github.com/microsoftdocs/mslearn-dotnet-kubernetes) to retrieve the code.

1. Open a command prompt or terminal window.
2. Open the root directory you want the code downloaded to. The code will be downloaded into a new folder in that location.
3. Run the following command to download, or clone, the sample repository.

BashCopy

git clone https://github.com/microsoftdocs/mslearn-dotnet-kubernetes

The code is downloaded into a new folder called **mslearn-dotnet-kubernetes**.

**Verify the Docker images by creating containers locally**

There are two containers in the Contoso Pizza Shop project. Before pushing the images to Docker Hub, let's use them to create the containers locally. After the containers are created and running, we will be able to browse the Contoso Pizza Company website and verify the microservices are running OK.

Follow these steps to create and run Docker containers from the Docker files you downloaded.

1. Make sure Docker Desktop is running.
2. Open a command prompt and move to the **mslearn-dotnet-kubernetes** directory.
3. Run the following command to build the containers.

BashCopy

docker-compose build

It may take a while to build the containers.

1. Run the following command to run the app and attach the containers.

BashCopy

docker-compose up

1. When the operation has finished, in a browser tab enter http://localhost:5902 to view the Contoso Pizza Shop menu.

**Sign in to Docker Hub**

The next step in uploading the images to Docker Hub is to sign into Docker Hub. From the command prompt, enter the following:

BashCopy

docker login

**Important**

Use the same username and password from when you created your Docker account. You can visit the [**Docker Hub website**](https://hub.docker.com/) to reset your password, if needed.

**Upload the images to Docker Hub**

1. Enter the following code to retag, or rename, the Docker images you created under your Docker username.

BashCopy

docker tag pizzafrontend [YOUR DOCKER USER NAME]/pizzafrontend

docker tag pizzabackend [YOUR DOCKER USER NAME]/pizzabackend

1. Then finally upload, or push, the Docker images to Docker Hub.

BashCopy

docker push [YOUR DOCKER USER NAME]/pizzafrontend

docker push [YOUR DOCKER USER NAME]/pizzabackend

In this exercise, you cloned Contoso Pizza Shop code from GitHub, used Dockerfiles contained within that code to create two Docker images and containers, and then pushed those images to Docker Hub.

Now you're ready to use Kubernetes to manage the deployment of Contoso Pizza Company's microservices.

**Exercise - Deploy a microservice container to Kubernetes**

Kubernetes runs containers for you. You describe what you want Kubernetes to do through a YAML file. This exercise will walk you through the creation of the file so you can deploy and run the **backend** service on Kubernetes.

**Important**

Before proceeding you must be sure you have a Kubernetes implementation installed. We will be using the implementation included with Docker Desktop. Follow [**these directions from Docker**](https://docs.docker.com/desktop/kubernetes/) in order to enable it.

**Create a deployment file for the backend service**

You can create a file to manage the deployment of a container into Kubernetes with a YAML file. Let's create a file to deploy the backend service.

1. Open a text editor, such as Visual Studio Code and switch to the directory you cloned the project files to earlier.
2. Create a new file in the root of the project called **backend-deploy.yml**.
3. Copy the following text into the file and then save it.

ymlCopy

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: pizzabackend

spec:

replicas: 1

template:

metadata:

labels:

app: pizzabackend

spec:

containers:

- name: pizzabackend

image: [YOUR DOCKER USER NAME]/pizzabackend:latest

ports:

- containerPort: 80

env:

- name: ASPNETCORE\_URLS

value: http://\*:80

selector:

matchLabels:

app: pizzabackend

---

apiVersion: v1

kind: Service

metadata:

name: pizzabackend

spec:

type: ClusterIP

ports:

- port: 80

selector:

app: pizzabackend

1. Replace the placeholder [YOUR DOCKER USER NAME] with your actual Docker username.

**Explaination:**

The YAML configuration provided describes two Kubernetes resources: a Deployment and a Service. Let's go through each section and understand their purpose:

Deployment:

apiVersion: Specifies the API version to use for the Deployment resource.

kind: Indicates the type of resource, which in this case is a Deployment.

metadata: Contains metadata about the Deployment, including its name.

spec: Defines the desired state and specifications for the Deployment.

replicas: Specifies the desired number of replicas for the Deployment, set to 1 in this case.

template: Describes the Pod template used for creating replicas.

metadata: Contains metadata specific to the Pod template.

labels: Labels used for identifying Pods associated with this Deployment.

spec: Specifies the specifications for the Pod template.

containers: Defines the container(s) to run within the Pod.

name: Specifies the name of the container.

image: Specifies the Docker image to use for the container.

ports: Specifies the ports to expose within the container.

containerPort: Specifies the port on which the container listens.

env: Specifies environment variables to be set within the container.

name: Specifies the name of the environment variable.

value: Specifies the value of the environment variable.

selector: Specifies the label selector used to match Pods to the Deployment.

Service:

apiVersion: Specifies the API version to use for the Service resource.

kind: Indicates the type of resource, which in this case is a Service.

metadata: Contains metadata about the Service, including its name.

spec: Defines the desired state and specifications for the Service.

type: Specifies the type of Service, which is ClusterIP in this case.

ports: Specifies the ports exposed by the Service.

port: Specifies the port number.

selector: Specifies the label selector used to route traffic to Pods associated with the Service.

This configuration creates a Deployment named "pizzabackend" with one replica and a Pod template that runs a container named "pizzabackend". The container uses a Docker image specified by [YOUR DOCKER USER NAME]/pizzabackend:latest, exposes port 80, and sets an environment variable ASPNETCORE\_URLS to http://\*:80. Additionally, a Service named "pizzabackend" is created, which routes traffic to Pods with the label app: pizzabackend and exposes port 80 internally within the cluster.

This file does a couple of things.

The first portion defines a deployment spec for the container that will be deployed into Kubernetes. It specifies there will be one replica, where to find the container image, which ports to open on the container, and sets some environment variables. This first portion also defines labels and names that the container and spec can be referenced by.

The second portion then defines that the container will run as a Kubernetes ClusterIP. For this module, you don't need to understand all of the specifics of ClusterIPs, but do know that this type of service doesn't expose an external IP address. It's only accessible from other services running from within the same Kubernetes cluster.

**Deploy and run the backend microservice**

Next let's deploy and run the microservice.

1. Open a command prompt to the same directory where you created the **backend-deploy.yml** file.
2. Run the following command.

BashCopy

**kubectl apply -f backend-deploy.yml**

This command is telling Kubernetes to run the file we created. It will download the image from Docker Hub and create the container.

Explanation:

The command **kubectl apply -f backend-deploy.yml** is used to apply the configuration defined in a YAML file (**backend-deploy.yml**) to create or update Kubernetes resources. Let's break down the command and understand its components:

* **kubectl**: It is the command-line tool used to interact with Kubernetes clusters.
* **apply**: This is a subcommand of **kubectl** used to apply configuration changes to Kubernetes resources.
* **-f backend-deploy.yml**: The **-f** flag specifies the file path or URL of the YAML file containing the resource configuration. In this case, it references a file named **backend-deploy.yml**.

By executing this command, Kubernetes will read the contents of the **backend-deploy.yml** file and apply the configuration specified within it. The YAML file typically includes definitions for one or more Kubernetes resources, such as deployments, services, or config maps.

The **apply** command performs the following actions:

1. If the resources described in the YAML file do not exist, Kubernetes will create them.
2. If the resources already exist, Kubernetes will update them based on the changes specified in the YAML file. It will modify the resources' properties to match the desired state defined in the YAML file.

Using **kubectl apply** with a YAML file allows you to manage and maintain the desired state of your Kubernetes resources. It simplifies the process of creating, updating, and configuring resources, ensuring consistency across deployments and environments.

1. The kubectl apply command will return quickly. But the container creation may take a while. To view the progress, use the following code.

BashCopy

kubectl get pods

In the resulting output, you'll see a row with **pizzabackend** followed by a string of random characters under the **NAME** column. When everything is ready, there'll be a **1/1** under the **READY** column and **Running** under the **STATUS** column.

The command **kubectl get pods** is used to retrieve information about the pods running within a Kubernetes cluster. Let's break down the command and understand its components:

* **kubectl**: It is the command-line tool used to interact with Kubernetes clusters.
* **get**: This is a subcommand of **kubectl** used to retrieve information about Kubernetes resources.
* **pods**: This argument specifies the resource type for which we want to retrieve information. In this case, it is "pods".

By executing the command **kubectl get pods**, you will receive a list of pods along with their status and other relevant details. The output typically includes information such as the name of the pod, its status (running, pending, completed), the node it is running on, the age of the pod, and the number of restarts.

Here is an example output:

sqlCopy code

NAME READY STATUS RESTARTS AGE pod1 1/1 Running 0 2m pod2 1/1 Running 0 1m

In this example, there are two pods named "pod1" and "pod2". Both pods are running, have successfully started without any restarts, and have been running for 2 minutes and 1 minute, respectively.

The **kubectl get pods** command is helpful for checking the status of pods, identifying any issues or errors, and monitoring the overall health of your applications running in the cluster.

Top of Form

1. Browse to http://localhost/pizzainfo. It will return an HTTP 404 Not Found message. This error is because the pizza backend service isn't accessible from the outside world.

**Create a deployment file and run the frontend service**

Much like the backend service, we'll need a deployment file for the front end as well.

1. Create a new file named **frontend-deploy.yml**
2. Paste the following code into the file.

ymlCopy

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: pizzafrontend

spec:

replicas: 1

template:

metadata:

labels:

app: pizzafrontend

spec:

containers:

- name: pizzafrontend

image: [YOUR DOCKER USER NAME]/pizzafrontend:latest

ports:

- containerPort: 80

env:

- name: ASPNETCORE\_URLS

value: http://\*:80

- name: backendUrl

value: http://pizzabackend

selector:

matchLabels:

app: pizzafrontend

---

apiVersion: v1

kind: Service

metadata:

name: pizzafrontend

spec:

type: LoadBalancer

ports:

- port: 80

selector:

app: pizzafrontend

1. Replace the placeholder [YOUR DOCKER USERNAME] with your actual Docker username.

You'll notice this file is similar to the one we created for the backend microservice. There are three differences:

* + We're specifying a different container to run under the deployment's spec.template.spec.containers.image value.
  + There's a new environment variable under the spec.template.spec.containers.env section. The code in the **pizzafrontend** application calls the backend, but because we haven't specified a fully qualified domain name nor will we know the IP address of the backend microservice we use the name we specified under the metadata.name node of the Deployment. Kubernetes will then take care of the rest.
  + And in the service section, we're specifying a value of **LoadBalancer** for spec.type. And port 80 is open. We'll now be able to browse the pizza frontend by navigating to **http://localhost**.

1. Deploy the container to Kubernetes with the following command.

BashCopy

kubectl apply -f frontend-deploy.yml

Again you can use kubectl get pods to see the status of the deployment. Once the row for **pizzafrontend** displays **Running** under the **STATUS** column, everything is ready to go.

1. When the container has been successfully deployed, browse to http://localhost to see both microservices running.

In this exercise, you created a deployment file that described exactly how you wanted the containers to run within Kubernetes. Then you had Kubernetes download the image from Docker Hub and start up the containers.

**Exercise - Scale a container instance in Kubernetes**

Your microservice may come under heavy load during certain times of the day. Kubernetes makes it easy to scale your microservice by adding additional instances for you.

1. Run the following command to scale the backend microservice to five instances.

BashCopy

kubectl scale --replicas=5 deployment/pizzabackend

The reason we need to specify **deployment/pizzabackend** instead of just **pizzabackend** is because we're scaling the entire Kubernetes deployment of the pizza backend service, and that will scale the instances of the individual pods correctly.

1. To verify five instances are up and running, run this command.

BashCopy

kubectl get pods

Once all the instances are spun up, you should see five pod instances (represented as individual rows) in the output. Each row will start with **pizzabackend** and then be followed by a random string.

1. To scale the instance back down, run the following command.

BashCopy

kubectl scale --replicas=1 deployment/pizzabackend