WA3007 Kubernetes for Developers

Student Labs

Web Age Solutions Inc.

Table of Contents

Environment

This course requires the following VM's [Virtual Machines] to be configured. You only need to run 1 VM at the time and you can shut down the VM that you are not using.

You should find shortcuts in the desktop named as the VM.

- **`VM_WA3007_REL_1_0`**
- **`WA2675-REL-2-1-minikube-MASTER`**

Lab 1 - Getting Started with Kubernetes

The main terminal-based tool for working with Kubernetes is the kubectl command. With it you can deploy objects like deployments, pods and services and see what is going on inside the cluster. Another important tool is a GUI web interface called the Kubernetes Dashboard. It provides much of the same functionality as kubectl but does so in a way that for some operations is easier to navigate and operate. In this lab we will take a look a t both of these tools and learn some of the things you can do with them.

1. kubectl Command
2. Kubernetes Dashboard

Part 1 - Setup

The following labs requires a VM named 'WA2675-REL-2-1-minikube-MASTER'

Close other VMs if running and double-click the shortcut on the desktop to launch the VM. **Use wasadmin for the user name and password**.

__1. Make sure that the VM is started and that you are logged in (wasadmin/wasadmin).

__2. Check that the internet is accessible from inside the VM.

__3. Open a new Terminal window by clicking **Applications > Terminal**.

__4. Verify the current user with the command 'whoami', it should be wasadmin.

__5. Navigate to the working directory **/home/wasadmin/Works**

__6. Check **t**he command prompt. It should be: **[wasadmin@localhost Works]$**

Part 2 - Start the Cluster

__1. Check if minikube is running:

```
minikube status
```

Output should say "Running" for each component and "Correctly Configured " for kubectl.

__2. If minikube is 'Stopped', start it with this command:

```
minikube start
```

It can take up several minutes for it to start up.

Wait for it to output the following line before moving on:

```
Done! kubectl is now configured to use "minikube"
```

If you repeatedly get the apiserver time out error message (ensure you have tried it at least 2-3 times), delete the old cluster by running the following command:

`minikube delete`

Once the delete is completed start minikube with this command which will also create a new cluster:

`minikube start -p minikube`

Wait for it to output the following line before moving on:

`Done! kubectl is now configured to use "minikube"`

The '**cached images**' error if it occurs can be ignored.

The '**Error processing tar file**' error if it occurs can be ignored.

__3. Check minikube status again with **`minikube status,`** it should indicate that minikube is running.

Part 3 - The kubectl command line interface (CLI)

The 'kubectl' command line interface is one of two main tolls available for you to configure and manage a Kubernetes cluster. The other tool is the Kubernetes Dashboard.

In this lab we will review a few of the high level kubectl resource commands you might expect to use on a daily basis including:

- explain
- create
- apply
- get
- delete

These commands work with most K8s resources like: **namespaces, nodes, pods, deployments, services, etc.**

Lets start by checking the cluster for any existing deployments. For this we will use the 'Get' command. The syntax of this command is

```
kubectl get {resource-type}
```

__1. Run the following **get** command:

```
kubectl get deployments
```

Your cluster may or may not show any existing deployments. If it does then the output would look something like this:

```
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
nginx-web   1/1     1            1           20h
```

If you have any existing deployments lets go ahead and delete them. You will use the kubectl delete command which has this syntax:

```
kubectl delete {resource-type} {resource-name}
```

__2. Run the following **delete** command. Make sure to substitute the name that was output from the earlier get command. Repeat this command for each deployment you found in your get listing.

```
kubectl delete deployments nginx-web
```

Another important command is **apply**. It is used to create and update resources and has the following syntax:

```
kubectl apply -f {resource-definition-filename}
```

__3. To use this command we'll need a resource definition file. Lets create one using the gedit editor:

```
gedit myapp.yaml
```

__4. Enter the following text into the file, then save it, and close the editor.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: apache
  name: apache
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: httpd
        image: httpd:latest
        ports:
        - containerPort: 80
```

But what do all of those settings mean? We can get a basic understanding by using the 'kubectl explain'command which has this syntax:

```
kubectl explain {resource-type}
```

__5. Try running the following explain command:

```
kubectl explain deployments
```

The output of the command is:

```
KIND:     Deployment
VERSION:  extensions/v1beta1

DESCRIPTION:
     DEPRECATED - This group version of Deployment is deprecated by
     apps/v1beta2/Deployment. See the release notes for more information.
     Deployment enables declarative updates for Pods and ReplicaSets.

FIELDS:
   apiVersion<string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest internal
     value, and may reject unrecognized values. More info:
     https://git.k8s.io/community/contributors/devel/api-conventions.md#resources

   kind     <string>
     Kind is a string value representing the REST resource this object
     represents. Servers may infer this from the endpoint the client submits
     requests to. Cannot be updated. In CamelCase. More info:
     https://git.k8s.io/community/contributors/devel/api-conventions.md#types-kinds
```

```
   metadata  <Object>
     Standard object metadata.

   spec      <Object>
     Specification of the desired behavior of the Deployment.

   status    <Object>
     Most recently observed status of the Deployment.
```

__6. Its also possible to drill down on this information. For example what about if we wanted to know more about the fields under **"metadata:"**? Try running the following command:

```
kubectl explain deployments.metadata
```

There are a lot of possible fields that could go under 'metadata' so the output of the above command is long. The first part of the output looks like this:

```
KIND:     Deployment
VERSION:  extensions/v1beta1

RESOURCE: metadata <Object>

DESCRIPTION:
     Standard object metadata.

     ObjectMeta is metadata that all persisted resources must have, which
     includes all objects users must create.

FIELDS:
   annotations            <map[string]string>
     Annotations is an unstructured key value map stored with a resource that
     may be set by external tools to store and retrieve arbitrary metadata. They
     are not queryable and should be preserved when modifying objects. More
     info: http://kubernetes.io/docs/user-guide/annotations

   clusterName            <string>
     The name of the cluster which the object belongs to. This is used to
   ...
```

Now that we better understand the deployment yaml file lets use it to create a deployment.

__7. Execute the following **apply** command to create a deployment based on the yaml file we just created:

```
kubectl apply -f myapp.yaml
```

__8. Check that the deployment was created:

```
kubectl get deployments
```

The deployment 'apache' should be in the list and the AVAILABLE field should say '1'. If if shows 0/1 wait a minute and run the command again until 1/1 is shown.

Another way to deploy an application is to use the **create** command. It is less flexible than **apply** but it does come in handy when you need to create a quick deployment.

__9. Deploy the '**alpine**' docker image using the following **create** command:

```
kubectl create deployment nginx --image nginx:latest
```

__10. Check deployments:

```
kubectl get deployments
```

The output should look like this:

```
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
apache      1/1     1            1           30m
nginx       1/1     1            1           46s
```

Next we will look at the kubernetes dashboard. Leave the deployments running.

Part 4 - Kubernetes Dashboard

Another API tool is the Kubernetes dashboard. The dashboard is a GUI web application that makes it easy to check how the cluster is working.

__1. Open a new terminal and run the following command to start the dashboard:

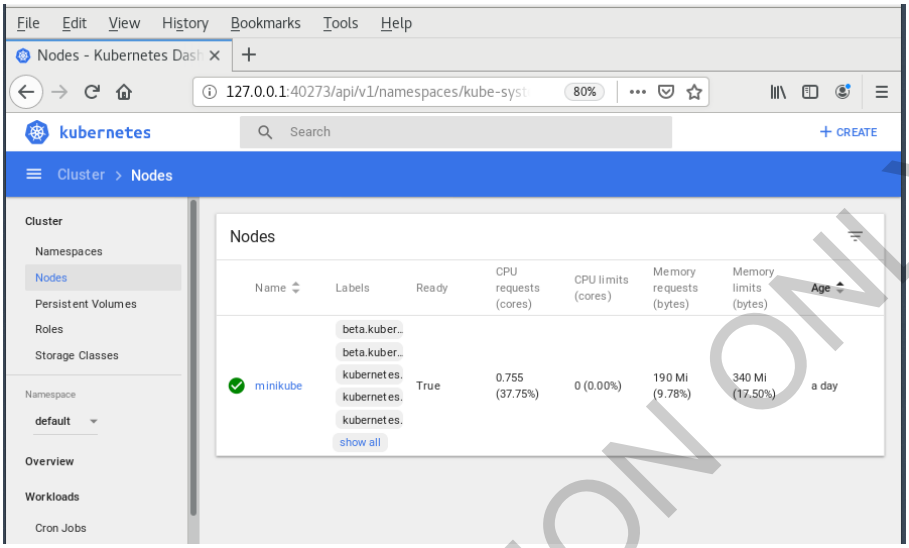
```
minikube dashboard
```

As it starts it should output the following to the console:

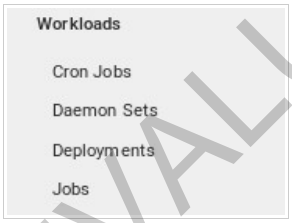
```
minikube dashboard
  Verifying dashboard health ...
  Launching proxy ...
  Verifying proxy health ...
  Opening http://127.0.0.1:40273/api/v1/namespaces/kube-
system/services/http:kubernetes-dashboard:/proxy/ in your default
browser...
```

If all goes well this will pop up a browser with the application's URL already loaded. If the browser does not pop up or the application does not get loaded in the browser go back to the output in the terminal, there you should see a URL. Copy the URL and paste it into a browser that is running in your VM. The app should then come up. If it still doesn't come up make sure you are not pasting the URL into a browser that is running outside the VM.

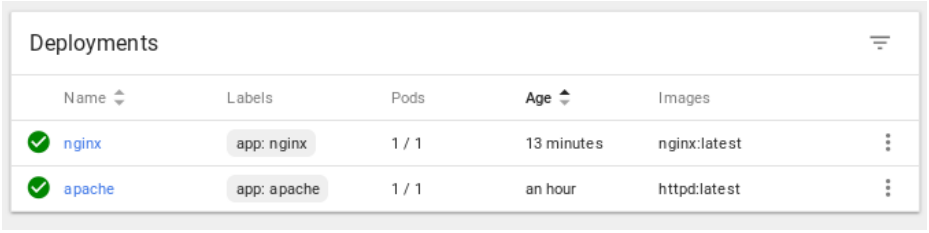
The dashboard app looks like this:



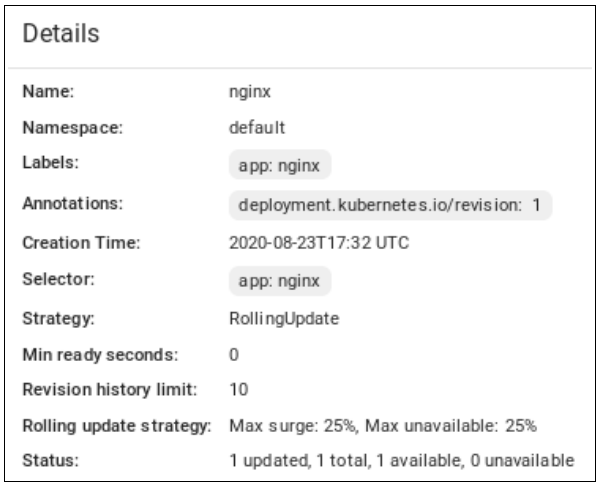
__2. The app has a navigation area to the left of the screen with links. Try clicking on the **deployments** link under **workloads**



__3. The main part of the screen should change to show your deployments:



__4. Drill down into the 'nginx' deployment by clicking on its name. This will show you the deployment details.



__5. Scroll down. You will see sections for related Replica Sets, Autoscalers and Events.

__6. Click on the **Edit** button at the upper right of the screen. This will pop up an edit screen with the deployments information in it. From this screen you can edit and apply updates to the deployment.

__7. Click cancel to dismiss the edit window.

__8. Other operations are also available at the top right of the window - Scale, Edit and Delete.

__9. Click 'deployments' under 'workloads' again from the navigation pane to the left. This will bring you back to the main deployments screen.

Feel free to try out some of the other links. As you become more familiar with what's available in Dashboard and as you work more with Kubernetes you may find some operations are easier to execute from the dashboard's gui than they are from a terminal using kubectl commands.

Part 5 - Review

In this lab we looked at two tools for interacting with Kubernetes clusters kubectl CLI and the Kubernetes Dashboard.

Lab 2 - Building a Docker Image with Dockerfile

Docker is an open-source containerization solution. Docker containers can run on any OS in an on-prem environment and also on any cloud platform. In this lab, you will create a custom Docker image by creating a Dockerfile.

Part 1 - Setup

The following labs requires a VM named 'VM_WA3007_REL_1_0'

Close other VMs if running and double-click the shortcut on the desktop to launch the VM. Use osboxes/osboxes.org for the user name and password.

__1. Make sure that the VM is started and that you are logged in (osboxes/osboxes.org).

__2. Check that the internet is accessible from inside the VM.

__3. Open a new Terminal window.

__4. Verify the current user with the command 'whoami', it should be **osboxes**.

__5. Switch to root with **osboxes.org** as the password.

```
sudo -i
```

__6. Create a directory to store project files and switch to the directory.

```
mkdir -p  /home/osboxes/Works && cd  /home/osboxes/Works
```

__7. Check **t**he command prompt. It should be: **root@osboxes:/home/osboxes/ Works#$**

Part 2 - Learning the Docker Command-line

Get quick information about Docker by running it without any arguments.

__1. Run the following command:

```
docker | less
```

__2. Navigate through the scrollable output using your arrow keys and review Docker's commands.

__3. Enter q to exit.

The commands list is shown below for you reference.

```
  attach      Attach local standard input, output, and error streams to a
  build       Build an image from a Dockerfile
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's
  events      Get real time events from the server
```

```
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  history     Show the history of an image
  images      List images
  import      Import the contents from a tarball to create a filesystem image
  info        Display system-wide information
  inspect     Return low-level information on Docker objects
  kill        Kill one or more running containers
  load        Load an image from a tar archive or STDIN
  login       Log in to a Docker registry
  logout      Log out from a Docker registry
  logs        Fetch the logs of a container
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  ps          List containers
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  rmi         Remove one or more images
  run         Run a command in a new container
  save        Save one or more images to a tar archive (streamed to STDOUT by
  search      Search the Docker Hub for images
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  version     Show the Docker version information
  wait        Block until one or more containers stop, then print their exit
```

__4. You can get command-specific help by using the **--help** flag added to the commands invocation line, e.g. to list containers created by Docker (the command is called **ps**), use the following command:

```
docker ps --help
```

More information on Docker's command-line tools can be obtained at ***https://docs.docker.com/reference/commandline/cli/***

__5. Enter the following command:

```
docker images
```

Note: This command displays docker images available on the machine.

Part 3 - Create Dockerfile for Building a Custom Image

In this part, you will create a Dockerfile which will build a custom Docker image.

__1. Extract the contents of a sample Java application.

```
unzip /LabFiles/sample-webapp.zip
```

__2. Switch to the sample-webapp directory.

```
cd sample-webapp
```

__3. In the terminal, type the following command to create Dockerfile:

```
gedit Dockerfile
```

__4. Type in following code:

```
FROM maven:3.6-jdk-8 AS builder
COPY . /app
WORKDIR /app
RUN mvn install

FROM openjdk:8-jdk-alpine
ARG TARGET=app/target
COPY --from=builder ${TARGET}/ /app/target
RUN ls /app/target
EXPOSE 8080
ENTRYPOINT ["java","-jar","./app/target/sample-webapp-1.0.jar"]
```

Note: The Dockerfile script is a 2-stage script. The first stage will obtain a base image from Docker Hub that will already have Maven and JDK8 preinstalled. After downloading the base image, your custom Java application source will be copied to it and Maven will build your application to geenerate the deployable artifact (.jar). The second stage will use a a very small base image that already has all the dependencies required to run your custom Java application.

__5. Click **Save** button.

__6. Close gedit. You will see some errors on the Terminal, ignore them.

__7. Type **ls** and verify **Dockerfile** new file is there.

__8. Run following command to build a custom image:

```
docker build -t sample-webapp:v1.0 .
```

This command builds/updates a custom image named dev-openjdk:v1.0. Don't forget to add the period at the end of docker build command.

Notice, Docker creates the custom image with JDK installed in it and the jar file deployed in the image. The first time you build the job, it will be slow since the image will get built for the first time. Subsequent runs will be faster since image will just get updated, not rebuilt from scratch.

Part 4 - Verify the Custom Image

In this part you will create a container based on the custom image you created in the previous part. You will also connect to the container, verify the jar file for the application exists, and execute the jar file.

__1. In the terminal, run following command to verify the custom image exists:

```
docker images | grep sample
```

Notice sample-app:v1.0 is there.

__2. Create a container based on the above image and connect to it:

```
docker run -d --name sample-webapp-container -p 8080:8080
sample-webapp:v1.0
```

Note: Ensure you run the above command as a single statement.

docker run is the short form of docker container run

-d runs the container in background as a daemon

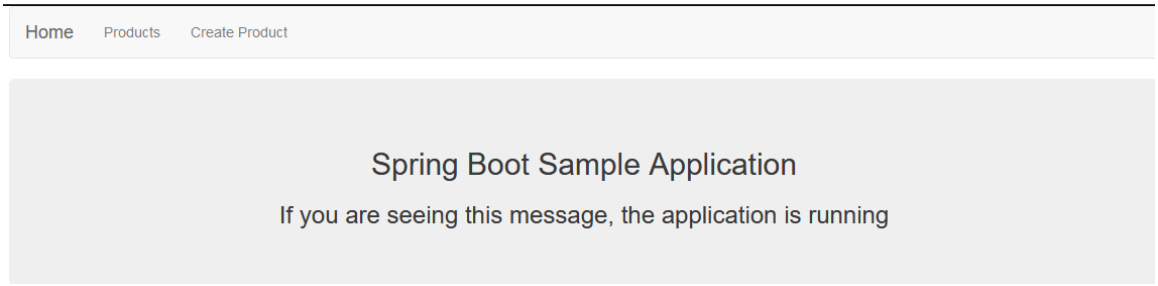
--name assigns a name to the container. You can create multiple containers based on the same image. Name can be utilized with commands to stop, start, and remove containers.

-p lets you perform port mapping. The number in front of the colon (:) symbol is where the port will be ava

__3. Open a web browser window and navigate to the following URL:

```
http://localhost:8080
```

__4. Verify the following page shows up



__5. If another service is already running on port 8080, you should remove the service and try the command again.

Part 5 - Interacting with the Container

In this part, you will check container logs and execute commands in the container

__1. Run the following command to check container logs.

```
docker logs sample-webapp-container
```

You can inspect logs to troubleshoot your application running in a container. If the application in a container crashes, the container will crash as well.

__2. Run the following command to inspect the Java version in the container.

```
docker exec -it sample-webapp-container java -version
```

__3. Run the following command to inspect the directory structure and verify your sample web application is deployed

```
docker exec -it sample-webapp-container ls app/target
```

Notice sample-webapp-1.0.jar is available in the container. ls app/target command got executed in the container. You could you cat or any other command to view contents of any file available in the container.

Part 6 - Stop and Delete the Container

In this part, you will stop and delete the docker container you created in the previous parts of the lab.

__1. Stop the container.

```
docker stop sample-webapp-container
```

Note: You can also obtain the container ID by using “docker ps -a” and use the ID instead of the container name.

__2. Destroy the container

```
docker rm sample-webapp-container
```

You can also delete an unused docker image by running “docker rmi <image_name>:<tag>

Part 7 - Review

In this lab, we reviewed the main Docker command-line operations, created a Dockerfile script, and ran it to build a Docker image to containerize a custom Java application.