Lab Guide

www.cognixia.com

COURSE NAME

Python

Setup Required:

Docker for Desktop [Recommeded]:

windows: https://download.docker.com/win/stable/Docker%20Desktop%20Installer.exe

Mac OS: https://download.docker.com/mac/stable/Docker.dmg

If not then Docker Tools + Minikube [The versions will vary]

Docker:

First command is to check the version:

```
docker version
```

sample output on mac:

```
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        afacb8b
 Built:             Wed Mar 11 01:21:11 2020
 OS/Arch:           darwin/amd64
 Experimental:      false
Server: Docker Engine - Community
 Engine:
  Version:          19.03.8
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.17
  Git commit:       afacb8b
  Built:            Wed Mar 11 01:29:16 2020
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          v1.2.13
  GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc:
  Version:          1.0.0-rc10
  GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
```

List containers

```
docker ps -a
```

```
CONTAINER ID        IMAGE                    COMMAND                  CREATED
STATUS                    PORTS                    NAMES
```

List images

```
docker images
```

sample output:

```
REPOSITORY                                                                    TAG
IMAGE ID            CREATED             SIZE
```

Obtain details information about your host

```
docker info
```

Gives you a lot of details about your system

Once you are done installing Docker, test your Docker installation by running the following:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
03f4658f8b78: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.
```

Running your first container

Now that you have everything setup, let's run an Alpine Linux container (a lightweight linux distribution) on your system and get a taste of the docker run command.

To get started, let's run the following in our terminal:

```
$ docker pull alpine
```

The `pull` command fetches the alpine **image** from the **Docker registry** and saves it in our system. You can use the `docker images` command to see a list of all images on your system.

```
$ docker images
```

```
REPOSITORY                TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
alpine                   latest              c51f86c28340        4 weeks ago
1.109 MB
hello-world               latest              690ed74de00f         5 months ago
960 B
```

Docker Run

Great! Let's now run a Docker **container** based on this image. To do that you are going to use the `docker run` command.

```
$ docker run alpine ls -l
total 48
drwxr-xr-x    2 root     root          4096 Mar  2 16:20 bin
drwxr-xr-x    5 root     root           360 Mar 18 09:47 dev
drwxr-xr-x   13 root     root          4096 Mar 18 09:47 etc
drwxr-xr-x    2 root     root          4096 Mar  2 16:20 home
drwxr-xr-x    5 root     root          4096 Mar  2 16:20 lib
......
......
```

What happened? Behind the scenes, a lot of stuff happened. When you call `run`,

1. The Docker client contacts the Docker daemon
2. The Docker daemon checks local store if the image (alpine in this case) is available locally, and if not, downloads it from Docker Store. (Since we have issued `docker pull alpine` before, the download step is not necessary)
3. The Docker daemon creates the container and then runs a command in that container.
4. The Docker daemon streams the output of the command to the Docker client

When you run `docker run alpine`, you provided a command (`ls -l`), so Docker started the command specified and you saw the listing.
Let's try something more exciting.

```
$ docker run alpine echo "hello from alpine"
hello from alpine
```

OK, that's some actual output. In this case, the Docker client dutifully ran the `echo` command in our alpine container and then exited it. If you've noticed, all of that happened pretty quickly. Imagine booting up a virtual machine, running a command and then killing it. Now you know why they say containers are fast!
Try another command.

```
$ docker run alpine /bin/sh
```

These interactive shells will exit after running any scripted commands, unless they are run in an interactive terminal - so for this example to not exit, you need to `docker run -it alpine /bin/sh`.

You are now inside the container shell and you can try out a few commands like `ls -l`, `uname -a` and others. Exit out of the container by giving the `exit` command.
Ok, now it's time to see the `docker ps` command. The `docker ps` command shows you all containers that are currently running.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

Since no containers are running, you see a blank line. Let's try a more useful variant: `docker ps -a`

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
36171a5da744        alpine              "/bin/sh"                5 minutes ago
Exited (0) 2 minutes ago                        fervent_newton
a6a9d46d0b2f        alpine             "echo 'hello from alp"    6 minutes ago
Exited (0) 6 minutes ago                        lonely_kilby
ff0a5c3750b9        alpine             "ls -l"                   8 minutes ago
Exited (0) 8 minutes ago                        elated_ramanujan
c317d0a9e3d2        hello-world         "/hello"                 34 seconds ago
Exited (0) 12 minutes ago                       stupefied_mcclintock
```

What you see above is a list of all containers that you ran. Notice that the `STATUS` column shows that these containers exited a few minutes ago. You're probably wondering if there is a way to run more than just one command in a container. Let's try that now:

```
$ docker run -it alpine /bin/sh
/ # ls
bin      dev      etc      home     lib      linuxrc  media    mnt      proc
root     run      sbin     sys      tmp      usr      var
/ # uname -a
Linux 97916e8cb5dc 4.4.27-moby #1 SMP Wed Oct 26 14:01:48 UTC 2016 x86_64 Linux
```

Running the `run` command with the `-it` flags attaches us to an interactive tty in the container. Now you can run as many commands in the container as you want. Take some time to run your favorite commands.

That concludes a whirlwind tour of the `docker run` command which would most likely be the command you'll use most often. It makes sense to spend some time getting comfortable with it. To find out more about `run`, use `docker run --help` to see a list of all flags it supports. As you proceed further, we'll see a few more variants of `docker run`.

Terminology

let's clarify some terminology that is used frequently in the Docker ecosystem.

- *Images* - The file system and configuration of our application which are used to create containers. To find out more about a Docker image, run `docker inspect alpine`. In the demo above, you used the `docker pull` command to download the **alpine** image. When you executed the command `docker run hello-world`, it also did a `docker pull` behind the scenes to download the **hello-world** image.
- *Containers* - Running instances of Docker images — containers run the actual applications. A container includes an application and all of its dependencies. It shares the kernel with other containers, and runs as an isolated process in user space on the host OS. You created a container using `docker run` which you did using the alpine image that you downloaded. A list of running containers can be seen using the `docker ps` command.
- *Docker daemon* - The background service running on the host that manages building, running and distributing Docker containers.
- *Docker client* - The command line tool that allows the user to interact with the Docker daemon.
- *Docker Store* - A registry of Docker images, where you can find trusted and enterprise ready containers, plugins, and Docker editions. You'll be using this later in this tutorial.

Deploy a Nginx server container

```
docker run -itd nginx:latest /bin/bash
```

Step 2.A : list the running container

```
docker ps
```

Step 2.b : Login to the container

```
docker  exec -it  <containerid> /bin/bash
```

Step 2.C : list process running in container

```
root@<containerid>:/# ps -ef
```

Step 3 : exit from the container

```
root@<containerid>:/# exit
```

Step 4 : Listing the Docker images

```
docker images
```

Step 5 : List the running containers

```
docker ps
```

Step 6 : Fetch info about container

```
docker inspect <containerid>
```

Step 8 : Login to your container

```
docker exec -i -t <containerid>/bin/bash
```

If you successful login you should be in your container bash login as below

```
root@<containerid>:/#
```

Step 9 : Validate your IP address of container

```
root@<containerid>:/#  ip a
```

Exiting the container

```
root@<containerid>:/#  exit
```

Another way to shutdown your container (login as root on your host)

```
docker inspect <containerid> | grep Pid

kill 4683
```

You will notice container is stopped

```
docker ps
```

Container Operations Part II:

Step 1 : List the running containers

```
docker ps
```

Step 2 : List all the containers

```
docker ps -a
```

Step 3 : Look at the CPU ,Memory and storage, network performance of container

```
docker stats <containerid>
```

Step 4 : Restarting the container

```
docker restart <containerid>
```

Step 5 : Pause the container

```
docker pause <containerid>
```

Step 6 : unPause the container

```
 docker unpause <containerid>
```

Step 7 : Rename your container Name

```
docker rename <containerid> web-server01
```

Syntax : docker rename <continerid> <desired name>

Step 8 : Stop your container

```
 docker stop <containerid>
```

Step 9 : Start your container

```
  docker start <containerid>
```

Step 10 : Delete the container

```
docker rm  <containerid>
```

Note : You cannot delete a container which is in start state

```
 docker stop <containerid>
```

Step 11 : Delete the container

```
docker rm  <containerid>
```

Step 12 : Validate that the container is deleted successfully

```
 docker ps -a
```

Volumes:

Step 1 : create a docker volume

Create and manage volumes

Unlike a bind mount, you can create and manage volumes outside the scope of any container.

Create a volume:

```
$ docker volume create data-volume
```

List volumes:

```
$ docker volume ls
```

Inspect a volume:

```
$ docker volume inspect data-volume
```

Remove a volume:

```
$ docker volume rm data-volume
```

Step 2 : create a container with volume

```
 docker run -d -it --name web-host -v myvol2:/data nginx:latest
```

////Notes

Myvol2 : is created on the host id does not exist

/data : is created in container if does not exist

```
docker volume ls

docker volume inspect myvol2
```

Step 3 : login to container and touch a file

```
$  docker exec -it web-host /bin/bash
```

Step 4 : create new new file in the /data directory and exit container

```
touch data/mydata.txt

ls data/

exit
```

Step 5 : **validate if file is there on your existing host file system**

```
ls -l /var/lib/docker/volumes/myvol2/_data
```

Step 6 : **remove the web-host container and create a new container**

```
docker  rm -f web-host

docker run -d -it --name new-host -v myvol2:/data nginx:latest
```

Step 7 : **login to container and touch a file**

```
docker exec -it new-host ls -l /data
```

`kubectl` uses a common syntax for all operations in the form of:

```
kubectl <command> <type> <name> <flags>
```

- command - The command or operation to perform. e.g. `apply`, `create`, `delete`, and `get`.
- type - The resource type or object.
- name - The name of the resource or object.
- flags - Optional flags to pass to the command.

Example:

```
$ kubectl create -f mypod.yaml

$ kubectl get pods

$ kubectl get pod mypod

$ kubectl delete pod mypod
```

Context and kubeconfig

`kubectl` allows a user to interact with and manage multiple Kubernetes clusters. To do this, it requires what is known as a context. A context consists of a combination of `cluster`, `namespace` and `user`.

- cluster - A friendly name, server address, and certificate for the Kubernetes cluster.
- namespace (optional) - The logical cluster or environment to use. If none is provided, it will use the default `default` namespace.
- user - The credentials used to connect to the cluster. This can be a combination of client certificate and key, username/password, or token.

These contexts are stored in a local yaml based config file referred to as the `kubeconfig`. For *nix based systems, the `kubeconfig` is stored in `$HOME/.kube/config` for Windows, it can be found in `%USERPROFILE%/.kube/config`

This config is viewable without having to view the file directly.

Command

```
$ kubectl config view
```

`kubectl config`

Managing all aspects of contexts is done via the `kubectl config` command. Some examples include:

- See the active context with `kubectl config current-context`.
- Get a list of available contexts with `kubectl config get-contexts`.
- Switch to using another context with the `kubectl config use-context <context-name>` command.
- Add a new context with `kubectl config set-context <context name> --cluster=<cluster name> --user=<user> --namespace=<namespace>`.

There can be quite a few specifics involved when adding a context, for the available options, please see the Configuring Multiple Clusters Kubernetes documentation.

Examples:

View the current contexts.

```
$ kubectl config get-contexts
```

View the current active context.

```
$ kubectl config current-context
```

Kubectl Basics

There are several `kubectl` commands that are frequently used for any sort of day-to-day operations. `get`, `create`, `apply`, `delete`, `describe`, and `logs`. Other commands can be listed simply with `kubectl --help`, or `kubectl <command> --help`.

`kubectl get`

`kubectl get` fetches and lists objects of a certain type or a specific object itself. It also supports outputting the information in several different useful formats including: json, yaml, wide (additional columns), or name (names only) via the `-o` or `--output` flag.

Command

```
kubectl get <type>

kubectl get <type> <name>

kubectl get <type> <name> -o <output format>
```

Examples

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    4h
kube-public   Active    4h
kube-system   Active    4h
$
$kubectl get pod mypod -o wide
NAME      READY     STATUS    RESTARTS   AGE       IP           NODE
mypod     1/1       Running   0          5m        172.17.0.6   minikube
```

---

kubectl create

`kubectl create` creates an object from the command line (`stdin`) or a supplied json/yaml manifest. The manifests can be specified with the `-f` or `--filename` flag that can point to either a file, or a directory containing multiple manifests.

Command

```
kubectl create <type> <parameters>
kubectl create -f <path to manifest>
```

Examples

```
$ kubectl create namespace dev
namespace "dev" created
$
$ kubectl create -f manifests/mypod.yaml
pod "mypod" created
```

`kubectl apply`

`kubectl apply` is similar to `kubectl create`. It will essentially update the resource if it is already created, or simply create it if does not yet exist. When it updates the config, it will save the previous version of it in an `annotation` on the created object itself.

WARNING: If the object was not created initially with `kubectl apply` it's updating behavior will act as a two-way diff. For more information on this, please see the kubectl apply documentation.

Just like `kubectl create` it takes a json or yaml manifest with the `-f` flag or accepts input from `stdin`.

Command

```
kubectl apply -f <path to manifest>
```

Examples

```
$ kubectl apply -f manifests/mypod.yaml

Warning: kubectl apply should be used on resource created by either kubectl
create --save-config or kubectl apply

pod "mypod" configured
```

`kubectl edit`

`kubectl edit` modifies a resource in place without having to apply an updated manifest. It fetches a copy of the desired object and opens it locally with the configured text editor, set by the `KUBE_EDITOR` or `EDITOR` Environment Variables. This command is useful for troubleshooting, but should be avoided in production scenarios as the changes will essentially be untracked.

Command

```
$ kubectl edit <type> <object name>
```

Examples

```
kubectl edit pod mypod
kubectl edit service myservice
```

---

kubectl delete

`kubectl delete` deletes the object from Kubernetes.

Command

```
kubectl delete <type> <name>
```

Examples

```
$ kubectl delete pod mypod
pod "mypod" deleted
```

---

kubectl describe

`kubectl describe` lists detailed information about the specific Kubernetes object. It is a very helpful troubleshooting tool.

Command

```
kubectl describe <type>
kubectl describe <type> <name>
```

kubectl logs

`kubectl logs` outputs the combined `stdout` and `stderr` logs from a pod. If more than one container exist in a `pod` the `-c` flag is used and the container name must be specified.

Command

```
kubectl logs <pod name>
kubectl logs <pod name> -c <container name>
```

Examples

```
$ kubectl logs mypod
172.17.0.1 - - [10/Mar/2018:18:14:15 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.57.0" "-"
172.17.0.1 - - [10/Mar/2018:18:14:17 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.57.0" "-"
```

Exercise: The Basics

Objective: Explore the basics. Create a namespace, a pod, then use the `kubectl` commands to describe and delete what was created.

1. Create the `dev` namespace.

```
kubectl create namespace dev
```

2. Apply the manifest `manifests/mypod.yaml`.

```
kubectl apply -f manifests/mypod.yaml
```

3. Get the yaml output of the created pod `mypod`.

```
kubectl get pod mypod -o yaml
```

4. Describe the pod `mypod`.

```
kubectl describe pod mypod
```

5. Clean up the pod by deleting it.

```
kubectl delete pod mypod
```

Summary: The `kubectl` *"CRUD"* commands are used frequently when interacting with a Kubernetes cluster. These simple tasks become 2nd nature as more experience is gained.

Accessing the Cluster

`kubectl` provides several mechanisms for accessing resources within the cluster remotely. For this tutorial, the focus will be on using `kubectl exec` to get a remote shell within a container, and `kubectl proxy` to gain access to the services exposed through the API proxy.

kubectl exec

`kubectl exec` executes a command within a Pod and can optionally spawn an interactive terminal within a remote container. When more than one container is present within a Pod, the `-c` or `--container` flag is required, followed by the container name.

If an interactive session is desired, the `-i` (`--stdin`) and `-t` (`--tty`) flags must be supplied.

Command

```
kubectl exec <pod name> -- <arg>

kubectl exec <pod name> -c <container name> -- <arg>

kubectl exec  -i -t <pod name> -c <container name> -- <arg>

kubectl exec  -it <pod name> -c <container name> -- <arg>
```

Example

```
$ kubectl exec mypod -c nginx -- printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=mypod
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
NGINX_VERSION=1.12.2
HOME=/root
$
$ kubectl exec -i -t mypod -c nginx -- /bin/sh
/ #
/ # cat /etc/alpine-release
3.5.2
```

kubectl proxy

`kubectl proxy` enables access to both the Kubernetes API-Server and to resources running within the cluster securely using `kubectl`. By default it creates a connection to the API-Server that can be accessed at `127.0.0.1:8001` or an alternative port by supplying the `-p` or `--port` flag.

Command

```
kubectl proxy
```

```
kubectl proxy --port=<port>
```

Examples

```
$ kubectl proxy &
Starting to serve on 127.0.0.1:8001

<from another terminal>
$ curl 127.0.0.1:8001/version
{
  "major": "",
  "minor": "",
  "gitVersion": "v1.9.0",
  "gitCommit": "925c127ec6b946659ad0fd596fa959be43f0cc05",
  "gitTreeState": "clean",
  "buildDate": "2018-01-26T19:04:38Z",
  "goVersion": "go1.9.1",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

The Kubernetes API-Server has the built-in capability to proxy to running services or pods within the cluster. This ability in conjunction with the `kubectl proxy` command allows a user to access those services or pods without having to expose them outside of the cluster.

```
http://<proxy_address>/api/v1/namespaces/<namespace>/<services|pod>/<servic
e_name|pod_name>[:port_name]/proxy
```

- proxy_address - The local proxy address - `127.0.0.1:8001`
- namespace - The namespace owning the resources to proxy to.
- service|pod - The type of resource you are trying to access, either `service` or `pod`.
- service_name|pod_name - The name of the `service` or `pod` to be accessed.
- [:port] - An optional port to proxy to. Will default to the first one exposed.

Example

```
http://127.0.0.1:8001/api/v1/namespaces/default/pods/mypod/proxy/
```

```
http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/kubernetes-
dashboard/proxy/
```

Kubectl Cheat Sheet: https://kubernetes.io/docs/reference/kubectl/cheatsheet/

Kubectl reference:

https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands

Namespaces

Namespaces are a logical cluster or environment. They are the primary method of partitioning a cluster or scoping access.

Exercise: Using Namespaces

Objectives: Learn how to create and switch between Kubernetes Namespaces using `kubectl`.

---

1. List the current namespaces

```
$ kubectl get namespaces
```

2. Create the `dev` namespace

```
$ kubectl create namespace dev
```

Summary: Namespaces function as the primary method of providing scoped names, access, and act as an umbrella for group based resource restriction. Creating and switching between them is quick and easy, but learning to use them is essential in the general usage of Kubernetes.

Pods

A pod is the atomic unit of Kubernetes. It is the smallest *"unit of work"* or *"management resource"* within the system and is the foundational building block of all Kubernetes Workloads.

Exercise: Creating Pods

Objective: Examine both single and multi-container Pods; including: viewing their attributes through the cli and their exposed Services through the API Server proxy.

---

1. Create a simple Pod called `pod-example` using the `nginx:stable-alpine` image and expose port `80`. Use the manifest `manifests/pod-example.yaml` or the yaml below.

manifests/pod-example.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  labels:
    app: nginx
    environment: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```

Command

```
$ kubectl create -f manifests/pod-example.yaml
```

2. Use `kubectl` to describe the Pod and note the available information.

```
$ kubectl describe pod pod-example
```

3. Use `kubectl proxy` to verify the web server running in the deployed Pod.

Command

```
$ kubectl proxy &
```

URL

http://127.0.0.1:8001/api/v1/namespaces/default/pods/pod-example/proxy/

The default "Welcome to nginx!" page should be visible.

4. Using the same steps as above, create a new Pod called `multi-container-example` using the manifest `manifests/pod-multi-container-example.yaml` or create a new one yourself with the below yaml.

manifests/pod-multi-container-example.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
  - name: content
    image: alpine:latest
    volumeMounts:
    - name: html
      mountPath: /html
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          echo $(date)"<br />" >> /html/index.html;
          sleep 5;
        done
  volumes:
  - name: html
    emptyDir: {}
```

Command

```
$ kubectl create -f manifests/pod-multi-container-example.yaml
```

Note: `spec.containers` is an array allowing you to use multiple containers within a Pod.

5. Use the proxy to verify the web server running in the deployed Pod.

Command

```
$ kubectl proxy
```

URL

```
http://127.0.0.1:8001/api/v1/namespaces/dev/pods/multi-container-
example/proxy/
```

There should be a repeating date-time-stamp.

Summary: Becoming familiar with creating and viewing the general aspects of a Pod is an important skill. While it is rare that one would manage Pods directly within Kubernetes, the knowledge of how to view, access and describe them is important and a common first-step in troubleshooting a possible Pod failure.

Labels and Selectors

Labels are key-value pairs that are used to identify, describe and group together related sets of objects or resources.

Selectors use labels to filter or select objects, and are used throughout Kubernetes.

https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Exercise: Using Labels and Selectors

Objective: Explore the methods of labeling objects in addition to filtering them with both equality and set-based selectors.

---

1. Label the Pod `pod-example` with `app=nginx` and `environment=dev` via `kubectl`.

```
$ kubectl label pod pod-example app=nginx environment=dev
```

2. View the labels with `kubectl` by passing the `--show-labels` flag

```
$ kubectl get pods --show-labels
```

3. Update the multi-container example manifest created previously with the labels `app=nginx` and `environment=prod` then apply it via `kubectl`.

manifests/pod-multi-container-example.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-example
  labels:
    app: nginx
    environment: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
  - name: content
    image: alpine:latest
    volumeMounts:
    - name: html
      mountPath: /html
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          date >> /html/index.html;
          sleep 5;
        done
  volumes:
  - name: html
    emptyDir: {}
```

Command

```
$ kubectl apply -f manifests/pod-multi-container-example.yaml
```

4. View the added labels with `kubectl` by passing the `--show-labels` flag once again.

```
$ kubectl get pods --show-labels
```

5. With the objects now labeled, use an equality based selector targeting the `prod` environment.

```
$ kubectl get pods --selector environment=prod
```

6. Do the same targeting the `nginx` app with the short version of the selector flag (`-l`).

```
$ kubectl get pods -l app=nginx
```

7. Use a set-based selector to view all pods where the `app` label is `nginx` and filter out any that are in the `prod` environment.

```
$ kubectl get pods -l 'app in (nginx), environment notin (prod)'
```

Summary: Kubernetes makes heavy use of labels and selectors in near every aspect of it. The usage of selectors may seem limited from the cli, but the concept can be extended to when it is used with higher level resources and objects.

Services

Services within Kubernetes are the unified method of accessing the exposed workloads of Pods. They are a durable resource (unlike Pods) that is given a static cluster-unique IP and provide simple load-balancing through kube-proxy.

Exercise: The clusterIP Service

Objective: Create a `ClusterIP` service and view the different ways it is accessible within the cluster.

---

1. Create `ClusterIP` service `clusterip` that targets Pods labeled with `app=nginx` forwarding port `80` using either the yaml below, or the manifest `manifests/service-clusterip.yaml`.

manifests/service-clusterip.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: clusterip
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Command

```
$ kubectl create -f manifests/service-clusterip.yaml
```

2. Describe the newly created service. Note the `IP` and the `Endpoints` fields.

```
$ kubectl describe service clusterip
```

3. View the service through `kube proxy` and refresh several times. It should serve up pages from both pods.

Command

```
$ kubectl proxy &
```

URL

```
http://127.0.0.1:8001/api/v1/namespaces/default/services/clusterip/proxy/
```

4. Lastly, verify that the generated DNS record has been created for the Service by using nslookup within the `example-pod` Pod that was provisioned in the above exercise.

```
$ kubectl exec pod-example -- nslookup clusterip.default.svc.cluster.local
```

```
#kubectl exec hello-world -- nslookup hello-
service.default.svc.cluster.local
```

It should return a valid response with the IP matching what was noted earlier when describing the Service.

Summary: The `ClusterIP` Service is the most commonly used Service within Kubernetes. Every `ClusterIP` Service is given a cluster unique IP and DNS name that maps to one or more Pod `Endpoints`. It functions as the main method in which exposed Pod Services are consumed within a Kubernetes Cluster.

Exercise: Using NodePort

Objective: Create a `NodePort` based Service and explore how it is available both inside and outside the cluster.

---

1. Create a `NodePort` Service called `nodeport` that targets Pods with the labels `app=nginx` and `environment=dev` forwarding port `80` in cluster, and port `32410` on the node itself. Use either the yaml below, or the manifest `manifests/service-nodeport.yaml`.

manifests/service-nodeport.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport
spec:
  type: NodePort
  selector:
    app: nginx
    environment: prod
  ports:
  - nodePort: 32410
    protocol: TCP
    port: 80
    targetPort: 80
```

Command

```
$ kubectl create -f manifests/service-nodeport.yaml
```

2. Describe the newly created Service Endpoint. Note the Service still has an internal cluster `IP`, and now additionally has a `NodePort`.

```
$ kubectl describe service nodeport
```

3. Open the newly exposed `nodeport` Service in a browser,

```
http://<<node_ip>>:32410    OR

http://<<master_ip>>:32410
```

4. Lastly, verify that the generated DNS record has been created for the Service by using nslookup within the `example-pod` Pod.

```
$ kubectl exec pod-example -- nslookup nodeport.default.svc.cluster.local
```

It should return a valid response with the IP matching what was noted earlier when describing the Service.

Summary: The `NodePort` Services extend the `ClusterIP` Service and additionally expose a port that is either statically defined, as above (port 32410) or dynamically taken from a range between 30000-32767. This port is then exposed on every node within the cluster and proxies to the created Service.

ReplicaSets

ReplicaSets are the primary method of managing Pod replicas and their lifecycle. This includes their scheduling, scaling, and deletion.

Their job is simple, always ensure the desired number of `replicas` that match the selector are running.

Exercise: Understanding ReplicaSets

Objective: Create and scale a ReplicaSet. Explore and gain an understanding of how the Pods are generated from the Pod template, and how they are targeted with selectors.

---

1. Begin by creating a ReplicaSet called `rs-example` with `3` `replicas`, using the `nginx:stable-alpine` image and configure the labels and selectors to target `app=nginx` and `env=prod`. The yaml block below or the manifest `manifests/rs-example.yaml` may be used.

manifests/rs-example.yaml

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  template:
    metadata:
      labels:
        app: nginx
        env: prod
    spec:
      containers:
      - name: nginx
        image: nginx:stable-alpine
        ports:
        - containerPort: 80
```

Command

```
$ kubectl create -f manifests/rs-example.yaml
```

2. Watch as the newly created ReplicaSet provisions the Pods based off the Pod Template.

```
$ kubectl get pods --watch --show-labels
```

Note that the newly provisioned Pods are given a name based off the ReplicaSet name appended with a 5 character random string. These Pods are labeled with the labels as specified in the manifest.

3. Scale ReplicaSet `rs-example` up to `5` replicas with the below command.

```
$ kubectl scale replicaset rs-example --replicas=5
```

Tip: `replicaset` can be substituted with `rs` when using `kubectl`.

4. Describe `rs-example` and take note of the `Replicas` and `Pod Status` field in addition to the `Events`.

```
$ kubectl describe rs rs-example
```

5. Now, using the `scale` command bring the replicas back down to `3`.

```
$ kubectl scale rs rs-example --replicas=3
```

6. Watch as the ReplicaSet Controller terminates 2 of the Pods to bring the cluster back into it's desired state of 3 replicas.

```
$ kubectl get pods --show-labels --watch
```

7. Once `rs-example` is back down to 3 Pods. Create an independent Pod manually with the same labels as the one targeted by `rs-example` from the manifest `manifests/pod-rs-example.yaml`.

manifests/pod-rs-example.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  labels:
    app: nginx
    env: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```

Command

```
$ kubectl create -f manifests/pod-rs-example.yaml
```

8. Immediately watch the Pods.

```
$ kubectl get pods --show-labels --watch
```

Note that the Pod is created and immediately terminated.

9. Describe `rs-example` and look at the `events`.

```
$ kubectl describe rs rs-example
```

There will be an entry with `Deleted pod: pod-example`. This is because a ReplicaSet targets ALL Pods matching the labels supplied in the selector.

Summary: ReplicaSets ensure a desired number of replicas matching the selector are present. They manage the lifecycle of ALL matching Pods. If the desired number of replicas matching the selector currently exist when the ReplicaSet is created, no new Pods will be created. If they are missing, then the ReplicaSet Controller will create new Pods based off the Pod Template till the desired number of Replicas are present.

Clean Up Command

```
kubectl delete rs rs-example
```

Deployments

Deployments are a declarative method of managing Pods via ReplicaSets. They provide rollback functionality in addition to more granular update control mechanisms.

Exercise: Using Deployments

Objective: Create, update and scale a Deployment as well as explore the relationship of Deployment, ReplicaSet and Pod.

---

1. Create a Deployment `deploy-example`. Configure it using the example yaml block below or use the manifest `manifests/deploy-example.yaml`. Additionally pass the `--record` flag to `kubectl` when you create the Deployment. The `--record` flag saves the command as an annotation, and it can be thought of similar to a git commit message.

manifests/deployment-example.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: deploy-example
spec:
  replicas: 3
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:stable-alpine
        ports:
        - containerPort: 80
```

Command

```
$ kubectl create -f manifests/deploy-example.yaml --record
```

2. Check the status of the Deployment.

```
$ kubectl get deployments
```

3. Once the Deployment is ready, view the current ReplicaSets and be sure to show the labels.

```
$ kubectl get rs --show-labels
```

Note the name and `pod-template-hash` label of the newly created ReplicaSet. The created ReplicaSet's name will include the `pod-template-hash`.

4. Describe the generated ReplicaSet.

```
$ kubectl describe rs deploy-example-<pod-template-hash>
```

Look at both the `Labels` and the `Selectors` fields. The `pod-template-hash` value has automatically been added to both the Labels and Selector of the ReplicaSet. Then take note of the `Controlled By` field. This will reference the direct parent object, and in this case the original `deploy-example` Deployment.

5. Now, get the Pods and pass the `--show-labels` flag.

```
$ kubectl get pods --show-labels
```

Just as with the ReplicaSet, the Pods name are labels include the `pod-template-hash`.

6. Describe one of the Pods.

```
$ kubectl describe pod deploy-example-<pod-template-hash-<random>
```

Look at the `Controlled By` field. It will contain a reference to the parent ReplicaSet, but not the parent Deployment.

Now that the relationship from Deployment to ReplicaSet to Pod is understood. It is time to update the `deploy-example` and see an update in action.

7. Update the `deploy-example` manifest and add a few additional labels to the Pod template. Once done, apply the change with the `--record` flag.

```
$ kubectl apply -f manifests/deploy-example.yaml --record
  < or >
$ kubectl edit deploy deploy-example --record
```

Tip: `deploy` can be substituted for `deployment` when using `kubectl`.

8. Immediately watch the Pods.

```
$ kubectl get pods --show-labels --watch
```

The old version of the Pods will be phased out one at a time and instances of the new version will take its place. The way in which this is controlled is through the `strategy` stanza. For specific documentation this feature, see the Deployment Strategy Documentation.

9. Now view the ReplicaSets.

```
$ kubectl get rs --show-labels
```

There will now be two ReplicaSets, with the previous version of the Deployment being scaled down to 0.

10. Now, scale the Deployment up as you would a ReplicaSet, and set the `replicas=5`.

```
$ kubectl scale deploy deploy-example --replicas=5
```

11. List the ReplicaSets.

```
$ kubectl get rs --show-labels
```

Note that there is NO new ReplicaSet generated. Scaling actions do NOT trigger a change in the Pod Template.

12. Just as before, describe the Deployment, ReplicaSet and one of the Pods. Note the `Events` and `Controlled By` fields. It should present a clear picture of relationship between objects during an update of a Deployment.

```
$ kubectl describe deploy deploy-example

$ kubectl describe rs deploy-example-<pod-template-hash>

$ kubectl describe pod deploy-example-<pod-template-hash-<random>
```

Summary: Deployments are the main method of managing applications deployed within Kubernetes. They create and supervise targeted ReplicaSets by generating a unique hash called the `pod-template-hash` and attaching it to child objects as a Label along with automatically including it in their Selector. This method of managing rollouts along with being able to define the methods and tolerances in the update strategy permits for a safe and seamless way of updating an application in place.

Exercise: Rolling Back a Deployment

Objective: Learn how to view the history of a Deployment and rollback to older revisions.

---

1. Use the `rollout` command to view the `history` of the Deployment `deploy-example`.

```
$ kubectl rollout history deployment deploy-example
```

There should be two revisions. One for when the Deployment was first created, and another when the additional Labels were added. The number of revisions saved is based off of the `revisionHistoryLimit` attribute in the Deployment spec.

2. Look at the details of a specific revision by passing the `--revision=<revision number>` flag.

```
$ kubectl rollout history deployment deploy-example --revision=1

$ kubectl rollout history deployment deploy-example --revision=2
```

Viewing the specific revision will display a summary of the Pod Template.

3. Choose to go back to revision `1` by using the `rollout undo` command.

```
$ kubectl rollout undo deployment deploy-example --to-revision=1
```

Tip: The `--to-revision` flag can be omitted if you wish to just go back to the previous configuration.

4. Immediately watch the Pods.

```
$ kubectl get pods --show-labels --watch
```

They will cycle through rolling back to the previous revision.

5. Describe the Deployment `deploy-example`.

```
$ kubectl describe deployment deploy-example
```

The events will describe the scaling back of the previous and switching over to the desired revision.

Summary: Understanding how to use `rollout` command to both get a diff of the different revisions as well as be able to roll-back to a previously known good configuration is an important aspect of Deployments that cannot be left out.

Clean Up Command

```
kubectl delete deploy deploy-example
```

DaemonSets

DaemonSets ensure that all nodes matching certain criteria will run an instance of the supplied Pod. They bypass default scheduling mechanisms and restrictions, and are ideal for cluster wide services such as log forwarding, or health monitoring.

Exercise: Managing DaemonSets

Objective: Experience creating, updating, and rolling back a DaemonSet. Additionally delve into the process of how they are scheduled and how an update occurs.

---

1. Create DaemonSet `ds-example` and pass the `--record` flag. Use the example yaml block below as a base, or use the manifest `manifests/ds-example.yaml` directly.

manifests/ds-example.yaml

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: ds-example
spec:
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        nodeType: edge
      containers:
      - name: nginx
        image: nginx:stable-alpine
        ports:
        - containerPort: 80
```

Command

```
$ kubectl create -f manifests/ds-example.yaml --record
```

2. View the current DaemonSets.

```
$ kubectl get daemonset
```

As there are no matching nodes, no Pods should be scheduled.

3. Label the `node01` node with `nodeType=edge`

```
$ kubectl label node <<your node>> nodeType=edge
```

4. View the current DaemonSets once again.

```
$ kubectl get daemonsets
```

There should now be a single instance of the DaemonSet `ds-example` deployed.

5. View the current Pods and display their labels with `--show-labels`.

```
$ kubectl get pods --show-labels
```

Note that the deployed Pod has a `controller-revision-hash` label. This is used like the `pod-template-hash` in a Deployment to track and allow for rollback functionality.

6. Describing the DaemonSet will provide you with status information regarding it's Deployment cluster wide.

```
$ kubectl describe ds ds-example
```

Tip: `ds` can be substituted for `daemonset` when using `kubectl`.

7. Update the DaemonSet by adding a few additional labels to the Pod Template and use the `--record` flag.

```
$ kubectl apply -f manifests/ds-example.yaml --record

  < or >

$ kubectl edit ds ds-example --record
```

8. Watch the Pods and be sure to show the labels.

```
$ kubectl get pods --show-labels --watch
```

The old version of the DaemonSet will be phased out one at a time and instances of the new version will take its place. Similar to Deployments, DaemonSets have their own equivalent to a Deployment's `strategy` in the form of `updateStrategy`. The defaults are generally suitable, but other tuning options may be set. For reference, see the Updating DaemonSet Documentation.

Summary: DaemonSets are usually used for important cluster-wide support services such as Pod Networking, Logging, or Monitoring. They differ from other workloads in that their scheduling bypasses normal mechanisms, and is centered around node placement. Like Deployments, they have their own `pod-template-hash` in the form of `controller-revision-hash` used for keeping track of Pod Template revisions and enabling rollback functionality.

Optional: Working with DaemonSet Revisions

Objective: Explore using the `rollout` command to rollback to a specific version of a DaemonSet.

1. Use the `rollout` command to view the `history` of the DaemonSet `ds-example`

```
$ kubectl rollout history ds ds-example
```

There should be two revisions. One for when the Deployment was first created, and another when the additional Labels were added. The number of revisions saved is based off of the `revisionHistoryLimit` attribute in the DaemonSet spec.

2. Look at the details of a specific revision by passing the `--revision=<revision number>` flag.

```
$ kubectl rollout history ds ds-example --revision=1
```

```
$ kubectl rollout history ds ds-example --revision=2
```

Viewing the specific revision will display the Pod Template.

3. Choose to go back to revision `1` by using the `rollout undo` command.

```
$ kubectl rollout undo ds ds-example --to-revision=1
```

Tip: The `--to-revision` flag can be omitted if you wish to just go back to the previous configuration.

4. Immediately watch the Pods.

```
$ kubectl get pods --show-labels --watch
```

They will cycle through rolling back to the previous revision.

5. Describe the DaemonSet `ds-example`.

```
$ kubectl describe ds ds-example
```

The events will be sparse with a single host, however in an actual Deployment they will describe the status of updating the DaemonSet cluster wide, cycling through hosts one-by-one.

Summary: Being able to use the `rollout` command with DaemonSets is import in scenarios where one may have to quickly go back to a previously known-good version. This becomes even more important for 'infrastructure' like services such as Pod Networking.

Clean Up Command

```
kubectl delete ds ds-example
```

Jobs and CronJobs

The Job Controller ensures one or more Pods are executed and successfully terminate. Essentially a task executor that can be run in parallel.

CronJobs are an extension of the Job Controller, and enable Jobs to be run on a schedule.

Exercise: Creating a Job

Objective: Create a Kubernetes `Job` and work to understand how the Pods are managed with `completions` and `parallelism` directives.

---

1. Create job `job-example` using the yaml below, or the manifest located at `manifests/job-example.yaml`

manifests/job-example.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
  backoffLimit: 4
  completions: 4
  parallelism: 2
  template:
    spec:
      containers:
      - name: hello
        image: alpine:latest
        command: ["/bin/sh", "-c"]
        args: ["echo hello from $HOSTNAME!"]
      restartPolicy: Never
```

Command

```
$ kubectl create -f manifests/job-example.yaml
```

2. Watch the Pods as they are being created.

```
$ kubectl get pods --show-labels --watch
```

Only two Pods are being provisioned at a time; adhering to the `parallelism` attribute. This is done until the total number of `completions` is satisfied. Additionally, the Pods are labeled with `controller-uid`, this acts as a unique ID for that specific Job.

When done, the Pods persist in a `Completed` state. They are not deleted after the Job is completed or failed. This is intentional to better support troubleshooting.

3. A summary of these events can be seen by describing the Job itself.

```
$ kubectl describe job job-example
```

4. Delete the job.

```
$ kubectl delete job job-example
```

5. View the Pods once more.

```
$ kubectl get pods
```

The Pods will now be deleted. They are cleaned up when the Job itself is removed.

Summary: Jobs are fire and forget one off tasks, batch processing or as an executor for a workflow engine. They *"run to completion"* or terminate gracefully adhering to the `completions` and `parallelism` directives.

Exercise: Scheduling a CronJob

Objective: Create a CronJob based off a Job Template. Understand how the Jobs are generated and how to suspend a job in the event of a problem.

---

1. Create CronJob `cronjob-example` based off the yaml below, or use the manifest `manifests/cronjob-example.yaml` It is configured to run the Job from the earlier example every minute, using the cron schedule `"*/1 * * * *"`. This schedule is UTC ONLY.

manifests/cronjob-example.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob-example
spec:
  schedule: "*/1 * * * *"
  successfulJobsHistoryLimit: 2
  failedJobsHistoryLimit: 1
  jobTemplate:
    spec:
      completions: 4
      parallelism: 2
      template:
        spec:
          containers:
          - name: hello
            image: alpine:latest
            command: ["/bin/sh", "-c"]
            args: ["echo hello from $HOSTNAME!"]
          restartPolicy: Never
```

Command

```
$ kubectl create -f manifests/cronjob-example.yaml
```

2. **Give it some time to run**, and then list the Jobs.

```
$ kubectl get jobs
```

There should be at least one Job named in the format `<cronjob-name>-<unix time stamp>`. Note the timestamp of the oldest Job.

3. Give it a few minutes and list the Jobs once again

```
$ kubectl get jobs
```

The oldest Job should have been removed. The CronJob controller will purge Jobs according to the `successfulJobHistoryLimit` and `failedJobHistoryLimit` attributes. In this case, it is retaining strictly the last 3 successful Jobs.

4. Describe the CronJob `cronjob-example`

```
$ kubectl describe CronJob cronjob-example
```

The events will show the records of the creation and deletion of the Jobs.

5. Edit the CronJob `cronjob-example` and locate the `Suspend` field. Then set it to true.

```
$ kubectl edit CronJob cronjob-example
```

This will prevent the cronjob from firing off any future events, and is useful to do to initially troubleshoot an issue without having to delete the CronJob directly.

6. Delete the CronJob

```
$ kubectl delete cronjob cronjob-example
```

Deleting the CronJob WILL delete all child Jobs. Use `Suspend` to *'stop'* the Job temporarily if attempting to troubleshoot.

Summary: CronJobs are a useful extension of Jobs. They are great for backup or other day-to-day tasks, with the only caveat being they adhere to a UTC ONLY schedule.

Clean Up Commands

```
kubectl delete CronJob cronjob-example
```

Volumes

Volumes within Kubernetes are storage that is tied to the Pod's lifecycle.A pod can have one or more type of volumes attached to it. These volumes are consumable by any of the containers within the pod. They can survive Pod restarts; however their durability beyond that is dependent on the Volume Type.

Exercise: Using Volumes with Pods

Objective: Understand how to add and reference volumes to a Pod and their containers.

---

1. Create a Pod with from the manifest `manifests/volume-example.yaml` or the yaml below.

manifests/volume-example.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
  - name: content
    image: alpine:latest
    volumeMounts:
    - name: html
      mountPath: /html
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          echo $(date)"<br />" >> /html/index.html;
          sleep 5;
        done
  volumes:
  - name: html
    emptyDir: {}
```

Command

```
$ kubectl create -f manifests/volume-example.yaml
```

Note the relationship between `volumes` in the Pod spec, and the `volumeMounts` directive in each container.

2. Exec into `content` container within the `volume-example` Pod, and `cat` the `html/index.html` file.

```
$ kubectl exec volume-example -c content -- /bin/sh -c "cat
/html/index.html"
```

You should see a list of date time-stamps. This is generated by the script being used as the entrypoint (`args`) of the content container.

3. Now do the same within the `nginx` container, using `cat` to see the content of `/usr/share/nginx/html/index.html` example.

```
$ kubectl exec volume-example -c nginx -- /bin/sh -c "cat
/usr/share/nginx/html/index.html"
```

You should see the same file.

4. Now try to append "nginx" to `index.html` from the `nginx` container.

```
$ kubectl exec volume-example -c nginx -- /bin/sh -c "echo nginx >>
/usr/share/nginx/html/index.html"
```

It should error out and complain about the file being read only. The `nginx` container has no reason to write to the file, and mounts the same Volume as read-only. Writing to the file is handled by the `content` container.

Summary: Pods may have multiple volumes using different Volume types. Those volumes in turn can be mounted to one or more containers within the Pod by adding them to the `volumeMounts` list. This is done by referencing their name and supplying their `mountPath`. Additionally, volumes may be mounted both read-write or read-only depending on the application, enabling a variety of use-cases.