Kubernetes

Table of Contents

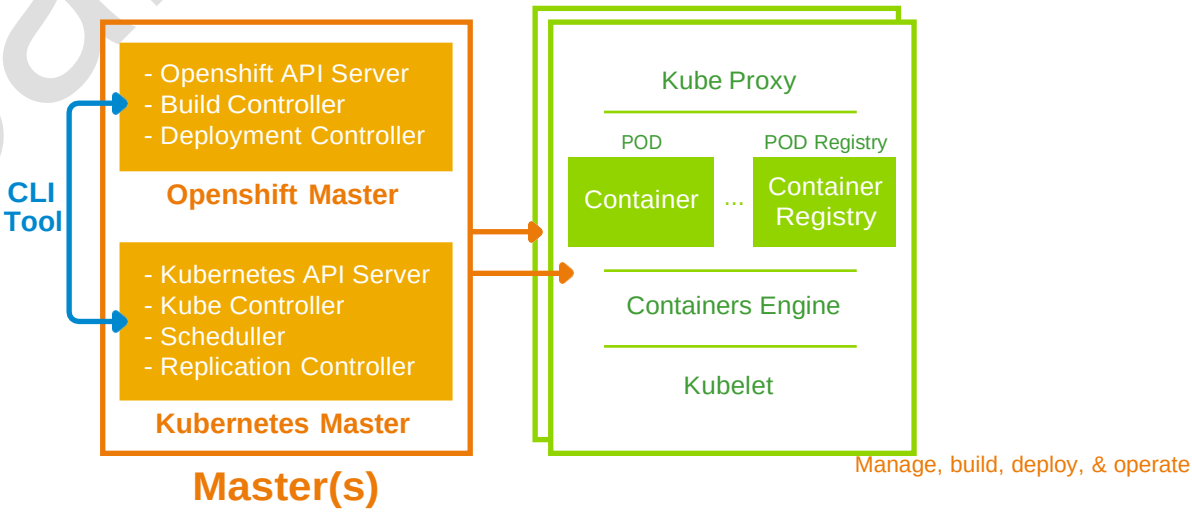
1. Introduction

Linux containers are a technology that allows you to package and isolate applications with their entire runtime environment–all of the files necessary to run. This makes it easy to move the contained application between environments (dev, test, production, etc.) while retaining full functionality.

Containers package applications with the files on which they depend. This reduces the friction between development and operations, simplifies application deployment, and accelerates delivery cycles—allowing you to deliver value to customers faster.

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application **containers** across clusters of hosts, providing container-centric infrastructure.

- Container orchestrator
- Runs Linux containers
  - Describe and launch containers
  - Monitors and maintains container state
  - Performs container-oriented networking



NODE(s)

Run containers and registries

2. Installation

2.A Installing `kubectl`

Download a pre-compiled release[1] and unzip it --- kubectl should be located in the platforms`/<os>/<arch>` directory.
[1] https://github.com/kubernetes/kubernetes/releases

Add kubectl to your path. Note, you can simply copy it into a directory that is already in your $PATH (e.g. /usr/local/bin). For example:

```
# Linux
$ sudo cp kubernetes/platforms/linux/amd64/kubectl /usr/local/bin/kubectl
# OS X
$ sudo cp kubernetes/platforms/darwin/amd64/kubectl /usr/local/bin/kubectl
```

You also need to ensure it's executable:

```
$ sudo chmod +x /usr/local/bin/kubectl
```

2.B Administration

To administer and interact with any given Kubernetes cluster (local or remote), you must set up your `kubeconfig` file. By default, kubectl configuration lives at `~/.kube/config`

You can also create a cluster in your local machine via Minikube (See section 3: Running Locally via Minikube)

```
current-context: federal-context
apiVersion: v1
clusters:
- cluster:
        api-version: v1
        server: http://cow.org:8080
  name: cow-cluster
- cluster:
        certificate-authority: path/to/my/cafile
        server: https://horse.org:4443
  name: horse-cluster
contexts:
- context:
        cluster: horse-cluster
        namespace: chisel-ns
        user: green-user
  name: federal-context
kind: Config
preferences:
  colors: true
users:
- name: green-user
  user:
        client-certificate: path/to/my/client/cert
        client-key: path/to/my/client/key
```

2.C You'll need more than Kubernetes:

Kubernetes operates at the application level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, logging, monitoring, etc.

However, Kubernetes is not an all-inclusive Platform as a Service (PaaS); therefore, you will still need to consider any needs for DevOps functionality separately:

- Networking
- Image registry
- Metrics and logging
- Complex deployments such as A/B and Blue/Green
- Application lifecycle management
- Application services such as database and messaging
- Self-service portal
- Container security

Much of this additional functionality is provided by the Red Hat OpenShift Container Platform (which includes Kubernetes.)

3. Running Locally via Minikube

Minikube is a tool that makes it easy to run Kubernetes locally --- it runs a single-node Kubernetes cluster inside a virtual machine on your laptop. This is useful for users looking to try out Kubernetes, or develop with it on a day-to-day basis.

3.A Prerequisites

Minikube requires that VT-x/AMD-v virtualization is enabled in BIOS on all platforms. For example:

```
# Linux
$ cat /proc/cpuinfo | grep 'vmx\|svm'
# OS X
$ sysctl -a | grep machdep.cpu.features | grep VMX
```

Make sure if the setting is enabled where this command should output something.

Install an x86 virtualization software package in your local machine:

- Linux: The latest VirtualBox
- OS X: The latest VirtualBox or VMware Fusion

3.B Install Minikube

Feel free to leave off the sudo mv minikube /usr/local/bin if you would like to add minikube to your path manually.

```
# Linux/
curl -Lo minikube https://storage.googleapis.com/minikube/releases/v0.12.2/minikube-linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/

# OS X
curl -Lo minikube https://storage.googleapis.com/minikube/releases/v0.12.2/minikube-darwin-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

3.C Install Kubectl

You will need to download and install the kubectl client binary to run commands against the cluster. For example:

```
# Linux/amd64
curl -Lo kubectl http://storage.googleapis.com/kubernetes-release/release/v1.3.0/bin/linux/amd64/kubectl && chmod +x kubectl && sudo mv kubectl /usr/local/bin/

# OS X/amd64
curl -Lo kubectl http://storage.googleapis.com/kubernetes-release/release/v1.3.0/bin/darwin/amd64/kubectl && chmod +x kubectl && sudo mv kubectl /usr/local/bin/
```

3.D Getting Started

Note that the IP below is dynamic and can change. It can be retrieved with minikube ip.

```
$ minikube start
Starting local Kubernetes cluster...
Running pre-create checks...
Creating machine...
Starting local Kubernetes cluster...

$ kubectl run hello-minikube --image=gcr.io/google_containers/echoserver:1.4
--port=8080
deployment "hello-minikube" created
$ kubectl expose deployment hello-minikube --type=NodePort
service "hello-minikube" exposed

# We have now launched an echoserver pod but we have to wait until the pod is up before curling/accessing it
# via the exposed service.
# To check whether the pod is up and running we can use the following:
$ kubectl get pod
NAME                              READY     STATUS              RESTARTS    AGE
hello-minikube-3383150820-vctvh   1/1       ContainerCreating   0           3s

# We can see that the pod is still being created from the ContainerCreating status
$ kubectl get pod
NAME                              READY     STATUS              RESTARTS    AGE
hello-minikube-3383150820-vctvh   1/1       Running             0           13s

# We can see that the pod is now Running and we will now be able to curl it:
$ curl $(minikube service hello-minikube --url)
CLIENT VALUES:
client_address=192.168.99.1
command=GET
real path=/
...
```

```
# To access the Kubernetes Dashboard, run this command in a shell after starting minikube to get the address:
$ minikube dashboard
$ minikube stop
Stopping local Kubernetes cluster...
Stopping "minikube"...
```

4. `kubectl` CLI

```
kubectl [command] [TYPE] [NAME] [flags]
```

- Command: Specifies the operation that you want to perform on one or more resources, for example create, get, delete.
- Type: Specifies the resource type. Resource types are case-sensitive and you can specify the singular, plural, or abbreviated forms.
- Name: Specifies the name of the resource. Names are case-sensitive. If the name is omitted, details for all resources are displayed.

4.A Kubectl Operations

All examples include the general syntax and description for kubectl operations:

Creating Objects

```
# example my-rc.yaml file for creating a object based Replication Controller
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80

# create resource(s)
$ kubectl create -f my-rc.yaml
replicationcontroller "nginx" created

# create resource(s) from url
$ kubectl create -f https://git.io/vPieo
pod "busybox0" created

# start a single instance of nginx
$ kubectl run nginx --image=nginx
deployment "nginx" created
```

```
# Get commands with basic output
$ kubectl get pod
NAME                                READY   STATUS    RESTARTS    AGE

busybox-sleep                       1/1     Running   0           8m

busybox-sleep-less                  1/1     Running   0           8m

busybox0                            1/1     Running   0           3m

hello-minikube-3015430129-vfgei     1/1     Running   0           20m

nginx-701339712-tkuma               1/1     Running   0           3m

# Get commands with yaml or json file format
$ kubectl get pod/nginx-cmpmt -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
        kubernetes.io/created-by: |

{"kind":"SerializedReference","apiVersion":"v1","reference":{"kind":"ReplicationControl
ler","namespace":"default","name":"nginx","uid":"01e01208-bb6a-11e6-a905-7eca61497d69",
"apiVersion":"v1","resourceVersion":"58757"}}
  creationTimestamp: 2016-12-06T04:11:05Z
  generateName: nginx-
  labels:
        app: nginx
  name: nginx-cmpmt
  namespace: default
  ownerReferences:
  - apiVersion: v1
        controller: true
        kind: ReplicationController
        name: nginx
        uid: 01e01208-bb6a-11e6-a905-7eca61497d69
  resourceVersion: "58815"
  selfLink: /api/v1/namespaces/default/pods/nginx-cmpmt
  uid: 01e10582-bb6a-11e6-a905-7eca61497d69
spec:
containers:
  - image: nginx
        imagePullPolicy: Always
        name: nginx
        ports:
        - containerPort: 80
        protocol: TCP
        resources: {}
        terminationMessagePath: /dev/termination-log
        volumeMounts:
        - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
        name: default-token-xxufg
        readOnly: true
  dnsPolicy: ClusterFirst
  nodeName: minikube
  restartPolicy: Always
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  volumes:
  - name: default-token-xxufg
```

```
        secret:
        secretName: default-token-xxufg
status:
  conditions:
  - lastProbeTime: null
        lastTransitionTime: 2016-12-06T04:11:05Z
        status: "True"
        type: Initialized
  - lastProbeTime: null
        lastTransitionTime: 2016-12-06T04:11:23Z
        status: "True"
        type: Ready
-   lastProbeTime: null
        lastTransitionTime: 2016-12-06T04:11:05Z
        status: "True"
        type: PodScheduled
containerStatuses:
  - containerID:
docker://46cdf4314702cc368cf76b46d690134bc78e0de313eb324409fefe088753ed78
        image: nginx
        imageID: docker://
sha256:abf312888d132e461c61484457ee9fd0125d666672e22f972f3b8c9a0ed3f0a1
        lastState: {}
        name: nginx
        ready: true
        restartCount: 0
        state:
        running:
        startedAt: 2016-12-06T04:11:23Z
  hostIP: 192.168.99.100
  phase: Running
  podIP: 172.17.0.13
  startTime: 2016-12-06T04:11:05Z

# Describe commands with verbose output
$ kubectl describe pods busybox-sleep
Name:           busybox-sleep
Namespace:    default
Node:           minikube/192.168.99.100
Start Time:    Sun, 27 Nov 2016 23:11:35 +0900
Labels:         <none>
Status:         Running
IP:             172.17.0.5
Controllers:    <none>
Containers:
  busybox:
        Container ID:
docker://4f599b509de0e8504b151e2dfeb98c14082ee149ec8da9132824e38095a6b86f
        Image:          busybox
        Image ID:       docker://
sha256:e02e811dd08fd49e7f6032625495118e63f597eb150403d02e3238af1df240ba
        Port:
        Args:
        sleep
        1000000
        State:                  Running
        Started:                Sun, 27 Nov 2016 23:11:43 +0900
        Ready:                  True
        Restart Count:          0
        Environment Variables:    <none>
Conditions:
  Type          Status
  Initialized     True
  Ready     True
  PodScheduled     True
```

```
Volumes:
  default-token-xxufg:
       Type:    Secret (a volume populated by a Secret)
       SecretName:    default-token-xxufg
QoS Tier:    BestEffort

# List Services Sorted by Name
$ kubectl get services --sort-by=.metadata.name
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
hello-minikube   10.0.0.38    <nodes>        8080/TCP   51m
kubernetes     10.0.0.1       <none>         443/TCP 53m

# Get ExternalIPs of all nodes
$ kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@.type=="ExternalIP")].
address}'
```

Viewing, Finding Resources

```
# Add a Label
$ kubectl label pods busybox-sleep new-label=new-busybox-sleep
pod "busybox-sleep" labeled

# Add an annotation
$ kubectl annotate pods busybox-sleep icon-url=http://goo.gl/XXBTWq
pod "busybox-sleep" annotated

# Auto scale a deployment "nginx"
$ kubectl autoscale deployment nginx --min=2 --max=5
deployment "nginx" autoscaled

# Rolling update pods of frontend-v1
$ kubectl rolling-update frontend-v1 -f frontend-v2.json

# Force replace, delete and then re-create the resource. Will cause a service outage
$ kubectl replace --force -f ./pod.json

# Create a service for a replicated nginx, which serves on port 80 and connects to the
containers on port 8000
$ kubectl expose rc nginx --port=80 --target-port=8000
```

Patching Resources

```
# Partially update a node
$ kubectl patch node k8s-node-1 -p '{"spec":{"unschedulable":true}}'
"k8s-node-1" patched

# Update a container's image; spec.containers[*].name is required because it's a merge key
$ kubectl patch pod valid-pod -p '{"spec":{"containers":[{"name":"kubernet
es-serve-hostname","image":"new image"}]}}'
"k8s-node-1" patched
```

Editing Resources

```
# Edit the service named docker-registry
$ kubectl edit svc/docker-registry
service "docker-registry" edited
```

Scaling Resources

```
# Scale a replicaset named nginx-701339712 to 5
$ kubectl scale --replicas=5 rs/nginx-701339712
replicaset "nginx-701339712" scaled

# Scale multiple replication controllers
$ kubectl scale --replicas=5 rc/foo rc/bar rc/baz
```

Deleting Resources

```
# Delete a pod using the type and specific name
$ kubectl delete pod/nginx-701339712-tkuma
pod "nginx-701339712-tkuma" deleted

# Delete pods and services with same names "baz" and "foo"
$ kubectl delete pod,service baz foo
pod "baz" deleted
service "foo" deleted

# Delete pods and services with label name=myLabel
$ kubectl delete pods,services -l name=myLabel

# Delete all pods and services in namespace my-ns
$ kubectl -n my-ns delete po,svc --all
```

Interacting with running pods

```
# dump pod logs (stdout)
$ kubectl logs busybox-sleep

# stream pod logs (stdout)
$ kubectl logs -f hello-minikube-3015430129-vfgei

# Run pod as interactive shell
$ kubectl run -i --tty busybox --image=busybox -- sh

# Attach to Running Container
$ kubectl attach my-pod -i

# Forward port to service
$ kubectl port-forward my-svc 6000
```

Interacting with running pods

```
# Mark a specific node as unschedulable
$ kubectl cordon minikube
node "minikube" cordoned

# Mark a specific  as schedulable
$ kubectl uncordon minikube
node "minikube" uncordoned

# Display addresses of the master and services
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/proxy/namespaces/
kube-system/services/kube-dns
kubernetes-dashboard is running at https://192.168.99.100:8443/api/v1/proxy/
namespaces/kube-system/services/kubernetes-dashboard
```

```
# Dump current cluster state to stdout
$ kubectl cluster-info dump

# Dump current cluster state to /path/to/cluster-state
$ kubectl cluster-info dump --output-directory=/path/to/cluster-state
```

`Docker` CLI & Dockerfile Cheat Sheet

Table of Contents

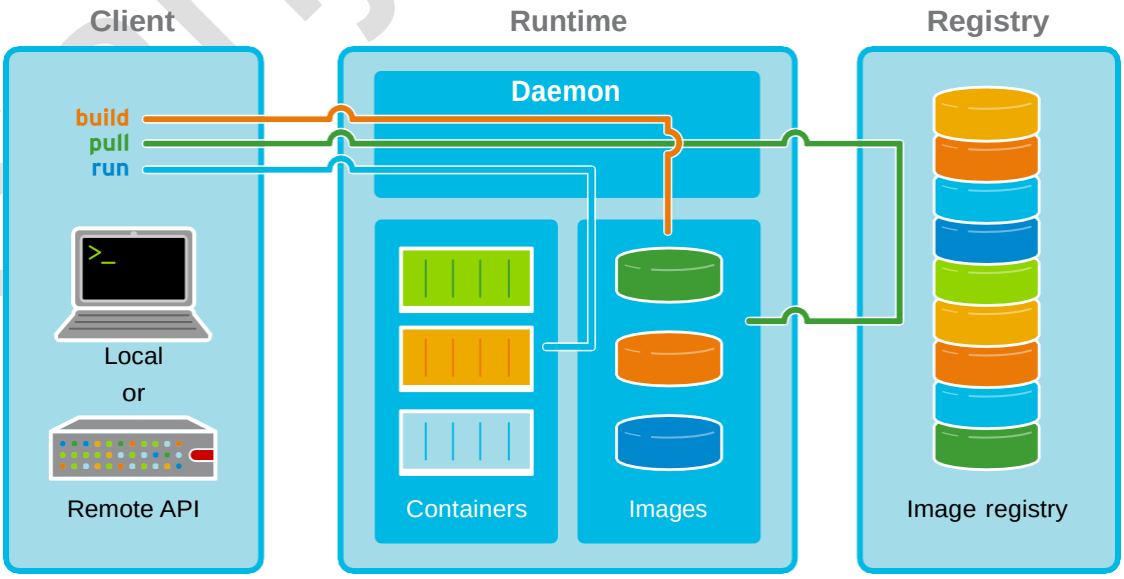
Introduction

Containers allow the packaging of your application (and everything that you need to run it) in a "container image". Inside a container you can include a base operating system, libraries, files and folders, environment variables, volume mount-points, and your application binaries.

A "container image" is a template for the execution of a container — It means that you can have multiple containers running from the same image, all sharing the same behavior, which promotes the scaling and distribution of the application. These images can be stored in a remote registry to ease the distribution.

Once a container is created, the execution is managed by the container runtime. You can interact with the container runtime through the "`docker`" command. The three primary components of a container architecture (client, runtime, & registry) are diagrammed below:

Container Architecture



1. `docker` CLI

1.1 Container Related Commands

`docker [CMD] [OPTS] [CONTAINER]`

Examples

All examples shown work in Red Hat Enterprise Linux

1. Run a container in interactive mode:

```
#Run a bash shell inside an image
$ docker run -it rhel7/rhel bash
#Check the release inside a container
[root@.../]# cat /etc/redhat-release
```

2. Run a container in detached mode:

```
$ docker run --name mywildfly -d -p 8080:8080 jboss/wildfly
```

3. Run a detached container in a previously created container network:

```
$ docker network create mynetwork
$ docker run --name mywildfly-net -d --net mynetwork \
      -p 8080:8080 jboss/wildfly
```

4. Run a detached container mounting a local folder inside the container:

```
$ docker run --name mywildfly-volume -d \
      -v myfolder/:/opt/jboss/wildfly/standalone/deployments/ \
      -p 8080:8080 jboss/wildflyjboss/wildfly
```

5. Follow the logs of a specific container:

```
$ docker logs -f mywildfly
$ docker logs -f [container-name|container-id]
```

6. List containers:

```
# List only active containers
$ docker ps
# List all containers
$ docker ps -a
```

7. Stop a container:

```
# Stop a container
$ docker stop [container-name|container-id]
# Stop a container (timeout = 1 second)
$ docker stop -t1
```

8. Remove a container:

```
# Remove a stopped container
$ docker rm [container-name|container-id]
# Force stop and remove a container
$ docker rm -f [container-name|container-id]
# Remove all containers
$ docker rm -f $(docker ps-aq)
# Remove all stopped containers
$ docker rm $(docker ps -q -f "status=exited")
```

9. Execute a new process in an existing container:

```
# Execute and access bash inside a WildFly container
$ docker exec -it mywildfly bash
```

Command	Description
`daemon`	Run the persistent process that manages containers
`attach`	Attach to a running container to view its ongoing output or to control it interactively
`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 on a container's filesystem
`exec`	Run a command in a running container
`export`	Export the contents of a container's filesystem as a tar archive
`kill`	Kill a running container using SIGKILL or a specified signal
`logs`	Fetch the logs of a container
`pause`	Pause all processes within a container
`port`	List port mappings, or look up the public-facing port that is NAT-ed to the PRIVATE_PORT
`ps`	List containers
`rename`	Rename a container
`restart`	Restart a container
`rm`	Remove one or more containers
`run`	Run a command in a new container
`start`	Start one or more containers
`stats`	Display one or more containers' resource usage statistics
`stop`	Stop a container by sending SIGTERM then SIGKILL after a grace period
`top`	Display the running processes of a container
`unpause`	Unpause all processes within a container
`update`	Update configuration of one or more containers
`wait`	Block until a container stops, then print its exit code

1.2 Image Related Commands

`docker [CMD] [OPTS] [IMAGE]`

Examples

All examples shown work in Red Hat Enterprise Linux

1. Build an image using a Dockerfile:

```
#Build an image
$ docker build -t [username/]<image-name>[:tag] <dockerfile-path>
#Build an image called myimage using the Dockerfile in the same folder where the command was executed
$ docker build -t myimage:latest .
```

2. Check the history of an image:

```
# Check the history of the jboss/wildfly image
$ docker history jboss/wildfly
# Check the history of an image
$ docker history [username/]<image-name>[:tag]
```

3: List the images:

```
$ docker images
```

4: Remove an image from the local registry:

```
$ docker rmi [username/]<image-name>[:tag]
```

5. Tag an image:

```
# Creates an image called “myimage” with the tag “v1” for the image jboss/wildfly:latest
$ docker tag jboss/wildfly myimage:v1
# Creates a new image with the latest tag
$ docker tag <image-name> <new-image-name>
# Creates a new image specifying the “new tag” from an existing image and tag
$ docker tag <image-name>[:tag][username/] <new-image-name>.[:new-tag]
```

6. Exporting and importing an image to an external file:

```
# Export the image to an external file
$ docker save -o <filename>.tar
# Import an image from an external file
$ docker load -i <filename>.tar
```

7 Push an image to a registry:

```
$ docker push [registry/][username/]<image-name>[:tag]
```

Command	Description
`build`	Build images from a Dockerfile
`history`	Show the history of an image
`images`	List images
`import`	Create an empty filesystem image and import the contents of the tarball into it
`info`	Display system-wide information
`inspect`	Return low-level information on a container or image
`load`	Load an image from a tar archive or STDIN
`pull`	Pull an image or a repository from the registry
`push`	Push an image or a repository to the registry
`rmi`	Remove one or more images
`save`	Save one or more images to a tar archive (streamed to STDOUT by default)
`search`	Search one or more configured container registries for images
`tag`	Tag an image into a repository

1.3 Network related commands

```
docker network [CMD] [OPTS]
```

Command	Description
`connect`	Connects a container to a network
`create`	Creates a new network with the specified name
`disconnect`	Disconnects a container from a network
`inspect`	Displays detailed information on a network
`ls`	Lists all the networks created by the user
`rm`	Deletes one or more networks

1.4 Network related commands

`Default is https://index.docker.io/v1/`

Command	Description
`login`	Log in to a container registry server. If no server is specified then default is used
`logout`	Log out from a container registry server. If no server is specified then default is used

1.5 Volume related commands

`docker volume [CMD] [OPTS]`

Command	Description
`create`	Create a volume
`inspect`	Return low-level information on a volume
`ls`	Lists volumes
`rm`	Remove a volume

1.6 Related commands

Command	Description
`events`	Get real time events from the server
`inspect`	Show version information

2. Dockerfile

The Dockerfile provides the instructions to build a container image through the `` `docker build -t [username/]<image-name>[:tag] <dockerfile-path>` `` command. It starts from a previously existing Base image (through the FROM clause) followed by any other needed Dockerfile instructions.

This process is very similar to a compilation of a source code into a binary output, but in this case the output of the Dockerfile will be a container image.

Example Dockerfile

This example creates a custom WildFly container with a custom administrative user. It also exposes the administrative port 9990 and binds the administrative interface publicly through the parameter 'bmanagement'.

```
# Use the existing WildFly image
FROM jboss/wildfly

# Add an administrative user
RUN /opt/jboss/wildfly/bin/add-user.sh admin Admin#70365 --silent

#Expose the administrative port
EXPOSE 8080 9990

#Bind the WildFly management to all IP addresses
CMD ["/opt/jboss/wildfly/bin/standalong.sh", "-b", "0.0.0.0",
"-bmanagement", "0.0.0.0"]
```

Using the example Dockerfile

```
# Build the WildFly image
$ docker build -t mywildfly .

#Run a WildFly server
$ docker run -it -p 8080:8080 -p 9990:9990 mywildfly

#Access the WildFly administrative console and log in with the credentials admin/Admin#70635
open http://<docker-daemon-ip>:9990 in a browser
```

Dockerfile instruction arguments

Command	Description
`FROM`	Sets the base image for subsequent
`MAINTAINER`	Sets the author field of the generated images
`RUN`	Execute commands in a new layer on top of the current image and commit the results
`CMD`	Allowed only once (if many then last one takes effect)
`LABEL`	Adds metadata to an image
`EXPOSE`	Informs container runtime that the container listens on the specified network ports at runtime
`ENV`	Sets an environment variable
`ADD`	Copy new files, directories, or remote file URLs from into the filesystem of the container
`COPY`	Copy new files or directories into the filesystem of the container
`ENTRYPOINT`	Allows you to configure a container that will run as an executable
`VOLUME`	Creates a mount point and marks it as holding externally mounted volumes from native host or other containers
`USER`	Sets the username or UID to use when running the image
`WORKDIR`	Sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, and `ADD` commands
`ARG`	Defines a variable that users can pass at build-time to the builder using `--build-arg`
`ONBUILD`	Adds an instruction to be executed later, when the image is used as the base for another build
`STOPSIGNAL`	Sets the system call signal that will be sent to the container to exit

Example: Running a web server container

```
$ mkdir -p www/                                  # Create a directory (if it doesn't already exist)

$ echo "Server is up" > www/index.html           # Make a text file to serve later

$ docker run -d \                                # Run process in a container as a daemon
  -p 8000:8000 \                                 # Map port 8000 in container to 8000 on host
  --name=pythonweb \                             # Name the container "pythonweb"
  -v `pwd`/www:/var/www/html \                   # Map container html to host www directory
  -w /var/www/html \                             # Set working directory to /var/www/html
  rhel7/rhel \                                   # Choose the rhel7/rhel directory
  /bin/python \                                  # Run the Python command for
  -m SimpleHTTPServer 8000                       #   a simple web server listening to port 8000

$ curl <container-daemon-ip>:8000                # Check that the server is working

$ docker ps                                      # See that the container is running
$ docker inspect pythonweb | less                # Inspect the container
$ docker exec -it pythonweb bash                 # Open the running container and look inside
```