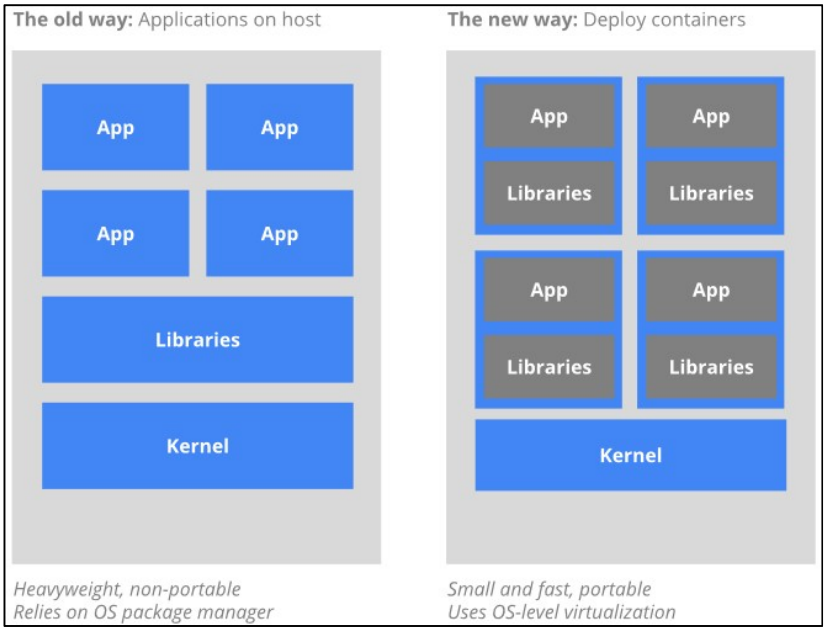
Table of Contents

Overview

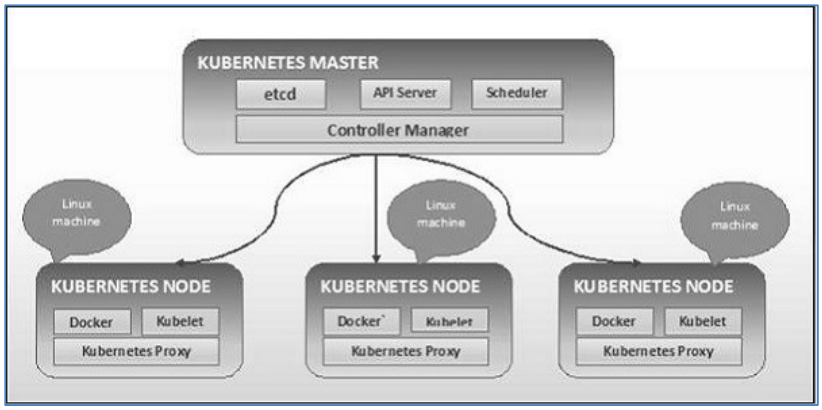
- Kubernetes is a
  - Portable,
  - Extensible,
  - Open-sourced platform
- for managing containerized workloads and services
- Google open-sourced K8s in 2014
- K8s features
  - Is a container platform
  - A microservices platform
  - A portable cloud platform
- Provides a container-centric management environment
- Orchestrates computing, networking and storage
- It is **not** a traditional, all-inclusive PaaS system
  - Since it operates at a container level rather than at the h/w level
- Why Containers?

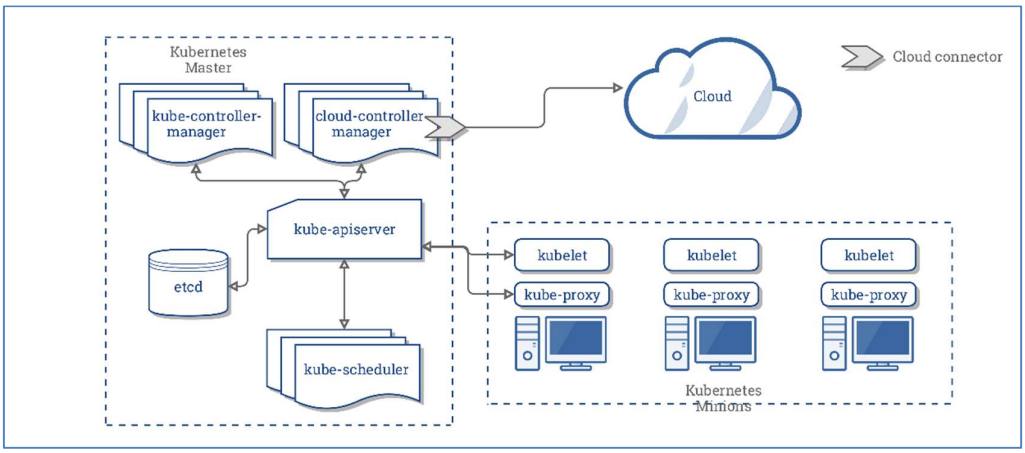


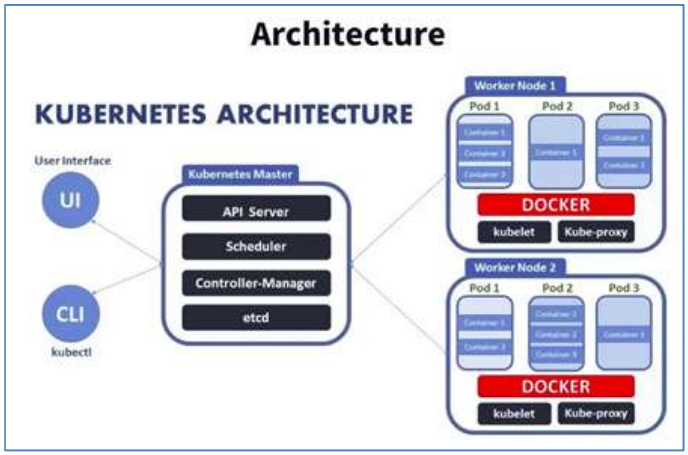
Kubernetes Architecture Overview

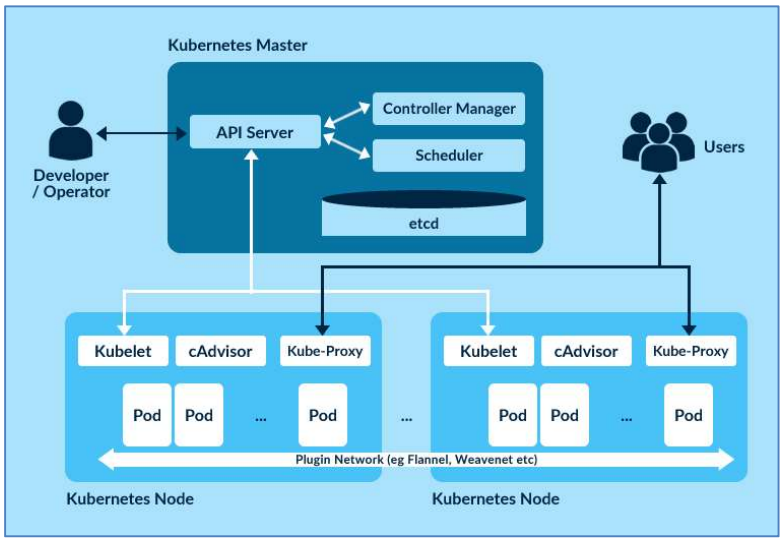
Kubernetes Cluster Archiercture

Kubernetes follows client-server architecture. Wherein, we have master installed on one machine and the node on separate Linux machines.

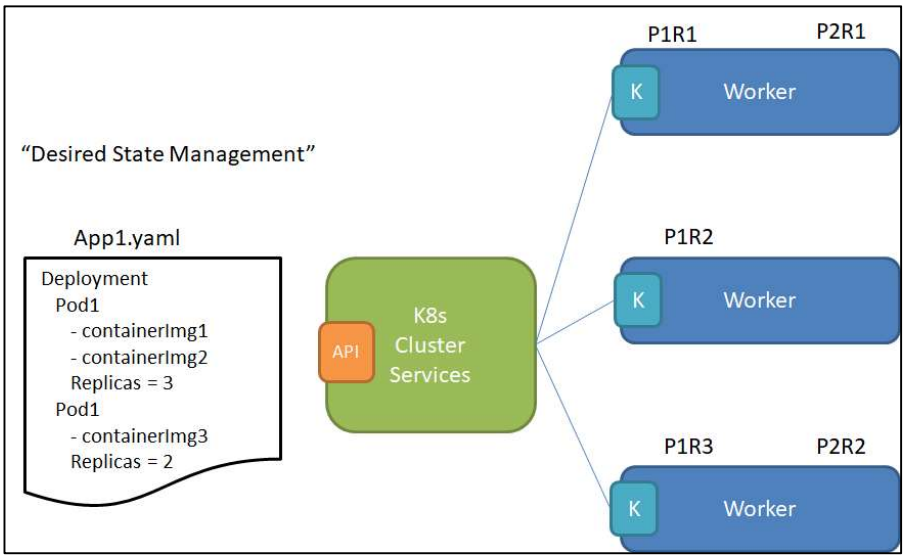

KUBERNETES MASTER
etcd
API Server
Scheduler
Controller Manager
Linux machine
Linux machine
Linux machine
KUBERNETES NODE
Docker
Kubelet
Kubernetes Proxy
KUBERNETES NODE
Docker
Kubelet
Kubernetes Proxy
KUBERNETES NODE
Docker
Kubelet
Kubernetes Proxy

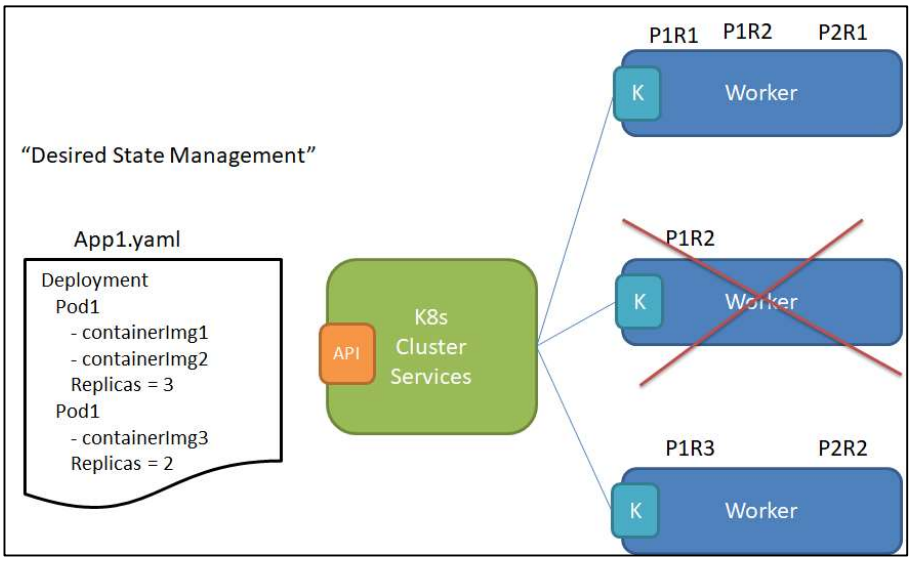

Kubernetes Master
kube-controller-manager
cloud-controller manager
Cloud
Cloud connector
kube-apiserver
etcd
kube-scheduler
kubelet
kube-proxy
kubelet
kube-proxy
kubelet
kube-proxy
Kubernetes Minions


Architecture
KUBERNETES ARCHITECTURE
User Interface
UI
CLI
kubectl
Kubernetes Master
API Server
Scheduler
Controller-Manager
etcd
Worker Node 1
Pod 1
Pod 2
Pod 3
DOCKER
kubelet
Kube-proxy
Worker Node 2
Pod 1
Pod 2
Pod 3
DOCKER
kubelet
Kube-proxy



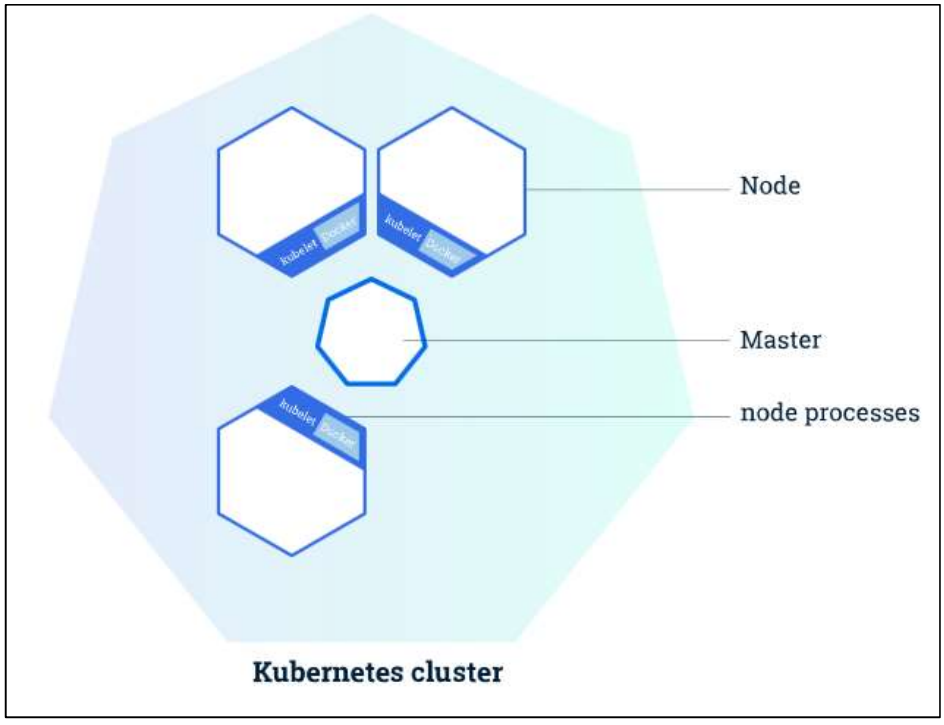
Desired State Management





Kubernetes Cluster

- Consists of two types of resources
  - Master: coordinates the cluster
  - Nodes: workers that run applications

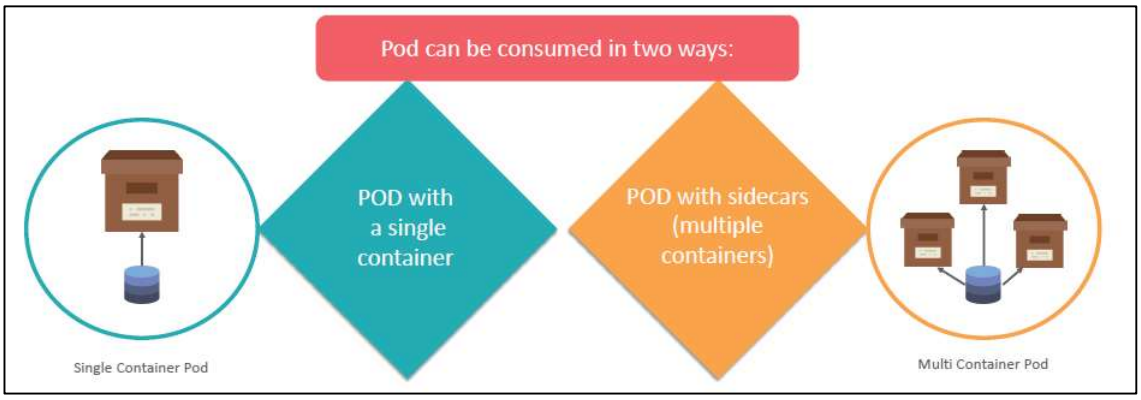


- Master
  - is responsible for managing the cluster
  - coordinates all activities in the cluster
    - scheduling apps
    - maintaining apps' desired state

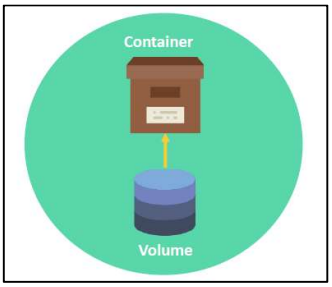
        - scaling apps
        - rolling out new updates
    - Node
        - Is a VM or physical server
        - aka as ***minions***
        - Serves as a worker machine in a k8s cluster
        - Each node has a Kubelet, which is an agent for managing the node and communicating with the k8s master
        - Should also have tools for handling container operations
            - Such as Docker
        - A k8s cluster that handles production traffic should have a minimum three nodes
- When you deploy an app on k8s,
    - You tell the master to start the app containers
    - The master schedules the containers to run on the cluster's nodes
    - The nodes communicate with the master using k8s API, which the master exposes

Pods

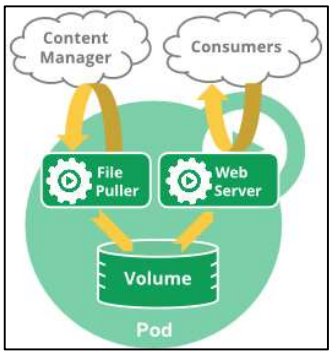
- A *Pod* is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy
- A *Pod* represents a unit of deployment in k8s cluster
- It is very easy to horizontally scale a Pod
- A *pod* (as in a pod of whales or pea pod) is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers
- It contains one or more application containers that are relatively tightly-coupled
- A Pod represents a running process on your cluster
- A Pod encapsulates an
    - application container (or, in some cases, multiple containers),
    - storage resources,
    - a unique network IP, and
    - options that govern how the container(s) should run
- A Pod represents a unit of deployment:
    - a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources
- ***Docker*** is the most common container runtime used in a Kubernetes Pod,
    - but Pods support other container runtimes as well
- Pods in a Kubernetes cluster can be used in two main ways:



- **Pods that run a single container**.
  - The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly



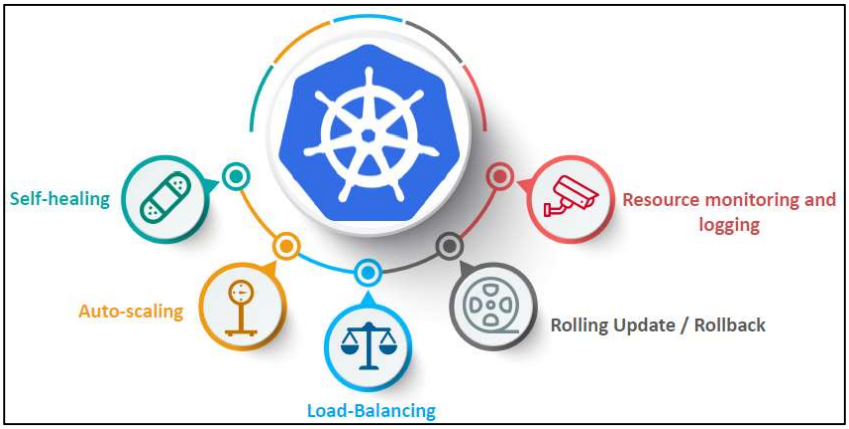
- **Pods that run multiple containers that need to work together**
  - Pod with side-cars (*multiple containers*)
  - A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources
  - These co-located containers might form a single cohesive unit of service– one container serving files from a shared volume to the public, while a separate "sidecar" container refreshes or updates those files
    - Second containers are generally called "***side-car***"
  - The Pod wraps these containers and storage resources together as a single manageable entity



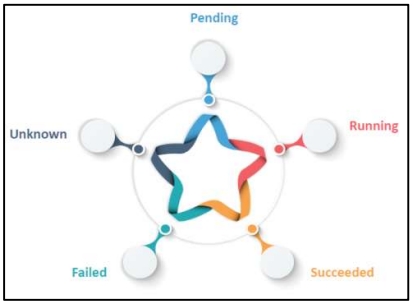
A pod diagram

- A multi-container pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers There are 2 containers that share the persistent volume
  - One of the containers is writing to the storage volume
  - The other container processes and publishes the same data to its consumers through a web server

- Pod features:



Pod Life-Cycle Phases



Pending	• It reflects the time spent in downloading the container images and creating them

	• It also means that system has accepted the Pod
Running	• Pod is tied-up with the node, and at least one container is running
Succeeded	• A container's termination in k8s was successful • It will not be restarted
Failed	• When one or more container's termination is unsuccessful • Termination due to failure is because of non-zero exit status of container
Unknown	• Where there is a communication problem of a container with the host machine, status of the container cannot be obtained, i.e.; *unknown*. • Since there is no status update, k8s system marks it as *unknown* • For such errors, communication channels should be checked first

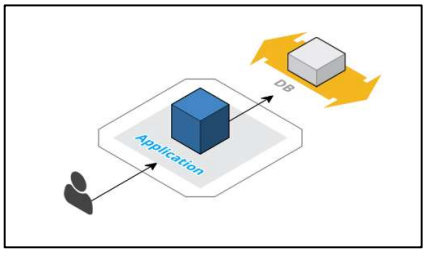
Kubernetes Node Status

Address	Hostname	Container engine provides the hostname, however you can override it and put a more meaningful hostname using –hostname-override parameter
	Internal IP Address	Internally routable IP address within the cluster (for internal communication only)
	External IP Address	Externally routable IP address to connect to/from outside the cluster
Condition	OutOfDisk	True, if there is insufficient disk space, otherwise false
	DiskPressure	True, if Disk capacity is low, else false
	MemoryPressure	True, if the node memory is slow, else false
	Networkunavailable	True, if network node is misconfigured, else false
	ConfigOK	True, if kubelet configuration is correct, else false
	Ready	True, if node is healthy False, if something is wrong Unknown, if nothing is heard from the node
Capacity	Describes the resources available on the node. Using this information, you can decide on the no. of pods that can be scheduled. These resources could be: • CPU • Memory • Storage	
Info	• Kernel details • O/S details • K8s details (like version no. etc.)	

Container Orchestration

Container Orchestration refers to the automated arrangement, coordination, and management of software containers.

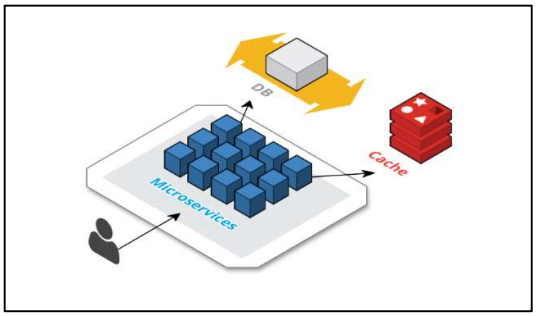
Why do we need this? Let's start with the following diagram:



If your current software infrastructure looks something like this — maybe Nginx/Apache + PHP/Python/Ruby/Node.js app running on a few containers that talk to a replicated DB — then you might not require container orchestration, you can probably manage everything yourself.

What if your application keeps growing? Let's say you keep adding more and more functionality until it becomes a massive monolith that is almost impossible to maintain and eats way too much CPU and RAM. You finally decide to split your application into smaller chunks, each responsible for one specific task, maintained by a team, aka. microservices.

Your infrastructure now kind of looks like this:



You now need a caching layer — maybe a queuing system as well — to increase performance, be able to process tasks asynchronously and quickly share data between the services. You also might want to run multiple instances of each microservice spanning multiple servers to make it highly available in a production environment…you see where I'm going with this.

You now have to think about challenges like:

- Service Discovery
- Load Balancing
- Secrets/configuration/storage management
- Health checks
- Auto-[scaling/restart/healing] of containers and nodes
- Zero-downtime deploys

This is where container orchestration platforms become extremely useful and powerful, because they offer a solution for most of those challenges.

So what choices do we have? Today, the main players are Kubernetes, AWS ECS and Docker Swarm, in order of popularity. Kubernetes has the largest community and is the most popular by a big margin (usage doubled in 2016, expected to 3–4x in 2017). There is also Kontena, which is much easier to setup compared to Kubernetes, but not as configurable and not very mature.

Volumes

A Kubernetes volume has an explicit lifetime - the same as the Pod that encloses it. Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts. Of course, when a Pod ceases to exist, the volume will cease to exist, too. Perhaps more importantly than this, Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously.

In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.

The concept of volume was present with the Docker, however the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost.

On the other hand, the volumes that are created through Kubernetes is not limited to any container. It supports any or all the containers deployed inside the pod of Kubernetes. A key advantage of Kubernetes volume is, it supports different kind of storage wherein the pod can use multiple of them at the same time.

Stateful vs Stateless Applications on Kubernetes

https://linuxhint.com/stateful-vs-stateless-kubernetes/

An important criteria to consider before running a new application, in production, is the app's underlying architecture. A term often used in this context is that the application is 'stateless' or that the application is 'stateful'. Both types have their own pros and cons. We will be having a Kubernetes cluster in the back of our mind when we talk about an application or a service running in production. You can install a Kubernetes cluster of your own on the cloud or you can have it running as a single node on your PC to get some practice with it.

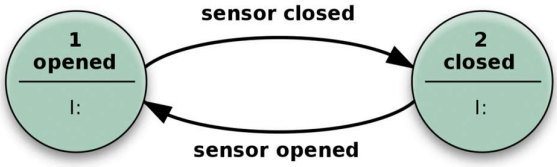
Let's start with a naïve definition of 'statelessness' and then slowly progress to a more rigorous and real-world view.

A stateless application is one which depends on no persistent storage. The only thing your cluster is responsible for is the code, and other static content, being hosted on it. That's it, no changing databases, no writes and no left over files when the pod is deleted.

A stateful application, on the other hand, has several other parameters it is supposed to look after in the cluster. There are dynamic databases which, even when the app is offline or deleted, persist on the disk. On a distributed system, like Kubernetes, this raises several issues. We will look at them in detail, but first let's clarify some misconceptions.

Stateless services aren't actually 'stateless'

What does it mean when we say the state of a system? Well, let's consider the following simple example of an automatic door.



The door opens when the sensor detects someone approaching, and it closes once the sensor gets no relevant input.

In practice, your stateless app is similar to this mechanism above. It can have many more states than just closed or open, and many different types of input as well making it more complex but essentially the same.

It can solve complicated problems by just receiving an input and performing actions which depend on both the input, and 'state' it is in. The number of possible states are predefined.

So statelessness is a misnomer.

Stateless applications, in practice, can also cheat a little by saving details about, say, the client sessions on the client itself (HTTP cookies are a great example) and still have a nice statelessness which would make them run flawlessly on the cluster.

For example, a client's session details like what products were saved in the cart and not checked out can all be stored on the client and the next time a session begins these relevant details are also recollected.

On a Kubernetes cluster, a stateless application has no persistent storage or volume associated with it. From an operations perspective, this is great news. Different pods all across the cluster can work independently with multiple requests coming to them simultaneously. If something goes wrong, you can just restart the application and it will go back to the initial state with little downtime.

Stateful services and the CAP theorem

The stateful services, on the other hand, will have to worry about lots and lots of edge-cases and weird issues. A pod is accompanied with at least one volume and if the data in that volume is corrupted then that persists even if the entire cluster gets rebooted.

For example, if you are running a database on a Kubernetes cluster, all the pods must have a local volume for storing the database. All of the data must be in perfect sync.

So if someone modifies an entry to the database, and that was done on pod A, and a read request comes on pod B to see that modified data, then pod B must show that latest data or give you an error message. This is known as consistency.

Consistency, in the context of a Kubernetes cluster, means *every read receives the most recent write or an error message*.

But this cuts against **availability**, one of the most important reasons for having a distributed system. Availability implies that your application functions as close to perfection as possbile, around the clock, with as little error as possible.

One may argue that you can avoid all of this if you have just one centralized database which is responsible for handling all of the persistent storage needs. Now we are back to having a single point of failure, which is yet another problem that a Kubernetes clusters is supposed to solve in the first place.

You need to have a decentralized way of storing persistent data in a cluster. Commonly referred to as network partitioning. Moreover, your cluster must be able to survive the failure of nodes running the stateful application. This is known as **partition tolerance**.

Any stateful service (or application), being run on a Kubernetes cluster, needs to have a balance between these three parameters. In the industry, it is known as the CAP theorem where the tradeoffs between Consistency and Availability are considered in presence of network Partitioning.

Kubernetes Core Concepts

Master node: Runs multiple controllers that are responsible for the health of the cluster, replication, scheduling, endpoints (linking *Services* and *Pods*), Kubernetes API, interacting with the underlying cloud providers etc. Generally it makes sure everything is running as it should be and looks after *worker nodes*.

Worker node (minion): Runs the Kubernetes agent that is responsible for running *Pod* containers via Docker or rkt, requests secrets or configurations, mounts required *Pod* volumes, does health checks and reports the status of *Pods* and the node to the rest of the system.

Pod: The smallest and simplest unit in the Kubernetes object model that you can create or deploy. It represents a running process in the cluster. Can contain one or multiple containers.

Deployment: Provides declarative updates for *Pods* (like the template for the *Pods*), for example the Docker image(s) to use, environment variables, how many *Pod* replicas to run, labels, node selectors, volumes etc.

DaemonSet: It's like a *Deployment* but instead runs a copy of a *Pod* (or multiple) on all (or some) nodes. Useful for things like log collection daemons (sumologic, fluentd), node monitoring daemons (datadog) and cluster storage daemons (glusterd).

ReplicaSet: Controller that ensures a specified number of *Pod* replicas (defined in the *Deployment*) is running at any given time.

Service: An abstraction which defines a logical set of *Pods* and a policyby which to access them (determined by a label selector). Generally it's used to expose Pods to other services within the cluster (using targetPort) or externally (using NodePort or LoadBalancer objects).

Kubernetes - Master Machine Components

etcd: It stores the configuration information which can be used by each of the nodes in the cluster. It is a high availability key value store that can be distributed among multiple nodes. It is accessible only by Kubernetes API server as it may have some sensitive information. It is a distributed key value Store which is accessible to all.

API Server: Kubernetes is an API server which provides all the operation on cluster using the API. API server implements an interface, which means different tools and libraries can readily communicate with it. Kubeconfig is a package along with the server side tools that can be used for communication. It exposes Kubernetes API.

Controller Manager: This component is responsible for most of the collectors that regulates the state of cluster and performs a task. In general, it can be considered as a daemon which runs in nonterminating loop and is responsible for collecting and sending information to API server. It works toward getting the shared state of cluster and then make changes to bring the current status of the server to the desired state. The key controllers are replication controller, endpoint controller,

namespace controller, and service account controller. The controller manager runs different kind of controllers to handle nodes, endpoints, etc.

Scheduler: This is one of the key components of Kubernetes master. It is a service in master responsible for distributing the workload. It is responsible for tracking utilization of working load on cluster nodes and then placing the workload on which resources are available and accept the workload. In other words, this is the mechanism responsible for allocating pods to available nodes. The scheduler is responsible for workload utilization and allocating pod to new node.

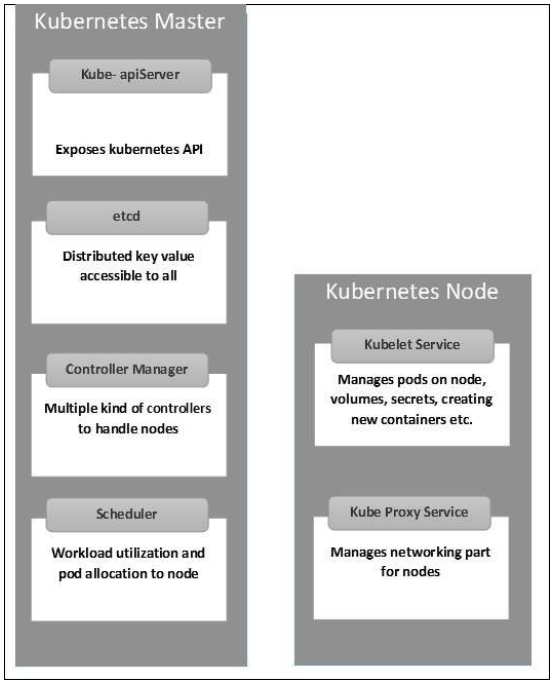
Kubernetes - Node Components

Docker: The first requirement of each node is Docker which helps in running the encapsulated application containers in a relatively isolated but lightweight operating environment.

Kubelet Service: This is a small service in each node responsible for relaying information to and from control plane service. It interacts with etcd store to read configuration details and wright values. This communicates with the master component to receive commands and work. The kubelet process then assumes responsibility for maintaining the state of work and the node server. It manages network rules, port forwarding, etc.

Kubernetes Proxy Service: This is a proxy service which runs on each node and helps in making services available to the external host. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing. It makes sure that the networking environment is predictable and accessible and at the same time it is isolated as well. It manages pods on node, volumes, secrets, creating new containers' health checkup, etc.

Kubernetes - Master and Node Structure



What is the Difference Between Kubernetes and Docker Swarm

Introduction

We have been coming across many container management engines, and while Kubernetes is the most popular container orchestration engine, Docker has Docker Swarm to do the same job and it easily integrates with Docker.

Kubernetes

Kubernetes is an open source system for managing containerized application in a clustered environment. Using Kubernetes in the right way helps the DevOps team to automatically scale an application up or down and update it with zero downtime.

Pros of Using Kubernetes

- **It's fast:** When it comes to continuously deploying new features without downtime, Kubernetes is a perfect choice. The goal of Kubernetes is to update an application with constant uptime. Its speed is measured through a number of features you can ship per hour while maintaining an available service.
- **Adheres to the principals of immutable infrastructure:** In a traditional way, if anything goes wrong with multiple updates, you don't have any record of how many updates you deployed and at which point the error occurred. In immutable infrastructure, if you wish to update an application, you need to build a container image with a new tag and deploy it, killing the old container with an old image version. In this way, you will have a record and get an insight of what you did and if there is any error, you can easily roll back to the previous image.
- **Provides declarative configuration:** Users can know in what state the system should be to avoid errors. Source control, unit tests, and other traditional tools can't be used with imperative configurations, but can be used with declarative configurations.
- **Deploy and update software at scale:** Scaling is easy due to the immutable, declarative nature of Kubernetes. Kubernetes offers several useful features for scaling purposes.
- **Horizontal Infrastructure Scaling:** Operations are done at the individual server level to apply horizontal scaling. The latest servers can be added or detached effortlessly.
- **Auto-scaling:** Based on the usage of CPU resources or other application metrics, you can change the number of containers that are running
- **Manual scaling:** You can manually scale the number of running containers through a command or the interface.
- **Replication controller:** The replication controller makes sure that the cluster has a specified number of equivalent pods in a running condition. If there are too many pods, a replication controller can remove extra pods or vice-versa.
- **Handles the availability of the application:** Kubernetes checks the health of nodes and containers as well as provides self-healing and auto-replacement if in-case pod crashes due

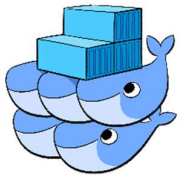
to an error. Moreover, it distributes the load across multiple pods to balance the resources quickly during accidental traffic.

- **Storage Volume:** In Kubernetes, data is shared across the containers, but if pods get killed volume is automatically removed. Moreover, data is stored remotely, if the pod is moved to another node, the data will remain until it is deleted by the user.

Cons of Using Kubernetes

- **Initial process takes time:** When a new process is created, you have to wait for the app to commence before it is available to the users. If you are migrating to Kubernetes, modifications in the codebase need to be done to make a start process more efficient so that users don't have a bad experience.
- **Migrating to stateless requires many efforts:** If your application is clustered or stateless, extra pods will not get configured and will have to rework on the configurations within your applications.
- **The installation process is tedious:** It is difficult to set up Kubernetes on your cluster if you are not using any cloud provider like Azure, Google or Amazon.

Docker Swarm



Docker Swarm is Docker's own native clustering solution for Docker containers which has an advantage of being tightly integrated into the ecosystem of Docker and uses its own API. It monitors the number of containers spread across clusters of servers and is the most convenient way to create clustered docker application without additional hardware. It provides you with a small-scale but useful orchestration system for the Dockerized app.

Pros of Using Docker Swarm

- **Runs at a faster pace:** When you were using a virtual environment, you may have realized that it takes a long time and includes the tedious procedure of booting up and starting the application that you want to run. With Docker Swarm, this is not a problem. Docker Swarm removes the need to boot up a full virtual machine and enables the app to run in a virtual and software-defined environment quickly and helps in DevOps implementation.
- **Documentation provides every bit of information:** The Docker team stands out when it comes to documentation! Docker is rapidly evolving and has received great applause for the entire platform. When a version gets released in a short interval of time, some platforms don't maintain/take care to maintain documentation. But Docker Swarm never compromises with it. If the information only applies to certain versions of a Docker Swarm, the documentation makes sure that all information is updated.
- **Provides simple and fast configuration:** One of the key benefits of Docker Swarm is that it simplifies matters. Docker Swarm enables the user to take their own configuration, put it

into a code and deploy it without any hassle. As Docker Swarm can be used in various environments, requirements are just not bound by the environment of the application.

- **Ensures that application is isolated:** Docker Swarm takes care that each container is isolated from the other containers and has its own resources. Various containers can be deployed for running the separate application in different stacks. Apart from this, Docker Swarm cleans app removal as each application runs on its own container. If the application is no longer required, you can delete its container. It won't leave any temporary or configuration files on your host OS.
- **Version control and component reuse:** With Docker Swarm, you can track consecutive versions of a container, examine differences or roll-back to the preceding versions. Containers reuse the components from the preceding layers which makes them noticeably lightweight.

Cons of Using Docker Swarm

- **Docker is platform-dependent:** Docker Swarm is a Linux-agonistic platform. Although Docker supports Windows and Mac OS X, it utilizes virtual machines to run on a non-Linux platform. An application which is designed to run in docker container on Windows can't run on Linux and vice versa.
- **Doesn't provide a storage option:** Docker Swarm doesn't provide a hassle-free way to connect containers to storage and this is one of the major disadvantages. Its data volumes require a lot of improvising on the host and manual configurations. If you're expecting Docker Swarm to solve the storage issues, it may get done but not in an efficient and user-friendly way.
- **Poor monitoring:** Docker Swarm provides the basic information about the container and if you are looking for a basic monitoring solution, then the stats command is suffice. If you are looking for an advanced monitoring than Docker Swarm is not an option. Although there are third-party tools available like CAdvisor which offers more monitoring, it is not feasible to collect more data about containers in real-time with Docker itself.

Docker and Kubernetes are Different; But not Rivals

As discussed earlier, Kubernetes and Docker both work at different levels but both can be used together. Kubernetes can be integrated with the Docker engine to carry out the scheduling and execution of Docker containers. As Docker and Kubernetes are both container orchestrators, they both can help to manage the number containers and also help in DevOps implementation. Both can automate most of the tasks that are involved in running containerized infrastructure and are open source software projects, governed by an Apache Licence 2.0. Apart from this, both use YAML-formatted files to govern how the tools orchestrate container clusters. When both of them are used together, both Docker and Kubernetes are the best tools for deploying modern cloud architecture. In the absence of Docker Swarm, both Kubernetes and Docker complement each other.

Kubernetes uses Docker as the main container engine solution and Docker recently announced that it can support Kubernetes as the orchestration layer of its enterprise edition. Apart from this, Docker approves certified Kubernetes program, which makes sure that all Kubernetes APIs functions as expected. Kubernetes uses the features of Docker Enterprise like Secure Image management, in

which Docker EE provides image scanning to check whether there is an issue in the image used in container. Another is Secure Automation in which organizations can remove inefficiencies such as scanning image for vulnerabilities.

Kubernetes or Docker: Which Is the Perfect Choice?

Use Kubernetes if:

- You are looking for a mature deployment and monitoring option.
- You are looking for fast and reliable response times.
- You are looking to develop a complex application and requires high resource computing without restrictions.
- You have a pretty big cluster.

Use Docker if,

- You are looking to initiate with the tool without spending much time on configuration and installation
- You are looking to develop a basic and standard application which is sufficient enough with default docker image
- Testing and running the same application on the different operating system is not an issue for you
- You want docker API experience and compatibility

Final Thoughts: Kubernetes and Docker As Friends

Whether you choose Kubernetes or Docker, both are considered the best and possess considerable differences. The best way to decide between the two of them is probably to consider which one you already know better or which one fits your existing software stack. If you need to develop the complex app, use Kubernetes, and if you are looking to develop the small-scale app, use Docker Swarm. Moreover, choosing the right one is a very comprehensive task and solely depends on your project requirements and target audience as well.

Installation

- Docker Desktop for Windows: https://docs.docker.com/docker-for-windows/install/
- Docker on Windows 10 Home:
  - Install Oracle Virtual Box and then create a new VM using docker-machine.
  - https://docs.docker.com/machine/get-started/
  - https://docs.docker.com/machine/drivers/virtualbox/
- Docker Desktop for Mac: https://docs.docker.com/docker-for-mac/install/
- Docker on Linux: https://runnable.com/docker/install-docker-on-linux

1. Open a command line window with administrator privileges.
2. Then use the chocolatey package manager to install minikube:
   a. Install Chocolatey from https://chocolatey.org/:

With cmd.exe as Admin:

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile -InputFormat None -ExecutionPolicy Bypass -Command "iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

With Powershell:

With PowerShell, there is an additional step. You must ensure Get-ExecutionPolicy is not Restricted. We suggest using Bypass to bypass the policy to get things installed or AllSigned for quite a bit more security.

- Run Get-ExecutionPolicy. If it returns Restricted, then run Set-ExecutionPolicy AllSigned or Set-ExecutionPolicy Bypass -Scope Process.
- Now run the following command: (copy command text)

```
Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

Additional considerations

NOTE: Please inspect https://chocolatey.org/install.ps1 prior to running any of these scripts to ensure safety. We already know it's safe, but you should verify the security and contents of ***any*** script from the internet you are not familiar with. All of these scripts download a remote PowerShell script and execute it on your machine.

Login for minikube VM is docker / tcuser

b. Then install minikube:

```
choco install minikube
```

3. Next install the kubernetes command line program with the chocolatey package manager:

```
choco install kubernetes-cli
```

Note: .minikube and .kube folders are in C:\Users\Ajay Singala

To Install minikube on HyperV:

i. Create an external Virtual Switch in Hyper-V Virtual Switch Manager (here, "myswitch")
ii. Run the following command to start the minikube VM with our applied changes:

```
minikube start --vm-driver hyperv --hyperv-virtual-switch myswitch
```

iii. Once the VM is running we will have two more steps to do to address a bug (Kube cluster shutsdown after a few minutes MiniKube VM still running) in minikube for Windows. We need to turn off Dynamic Memory for the minikube VM:
   a. Run:

```
minikube stop --alsologtostderr
```

   b. In HyperV Manager, disable "Enable Dynamic Memory" for the minikube VM
   c. Then, run:

```
minikube start
```

To Install minikube on VirtualBox:

i. Make sure Hyper-V is disabled
ii. Run:

```
minikube.exe start --vm-driver=virtualbox
```

4. Confirm the installation is ready

```
kubectl get pods -n kube-system
```

5. Or Run to open the dashboard in a browser:

```
minikube dashboard
```

Enable Kubernetes on Docker and Enable Dashboard

http://collabnix.com/kubernetes-dashboard-on-docker-desktop-for-windows-2-0-0-3-in-2-minutes/

In Docker settings, click on Kubernetes and select checkboxes for Enable Kubernetes, Deploy Docker Stacks to K8s by default and Show system containers (advanced). k8s will start and be running in some time.

Verify:

```
kubectl version
```

Setup the k8s Dashboard

```
kubectl version

kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.1/src/deploy/
recommended/kubernetes-dashboard.yaml

kubectl proxy
```

Open the following in your browser:

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

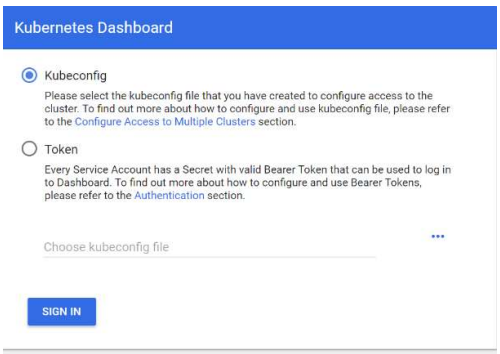
You will get:

```
"status": "Failure",
"message": "no endpoints available for service \"Kubernetes-dashboard\"",
"reason": "ServiceUnavailable",
"code": 503
```

Open this url in your browser:

```
http://localhost:8001/api/v1/namespaces/kube-
system/services/https:kubernetes-dashboard:/proxy/#!/login
```

Will show the following:



Enter the following commands on the terminal window:

```
$TOKEN=((kubectl -n kube-system describe secret default | Select-String
"token:") -split " +")[1]
```

```
kubectl config set-credentials docker-for-desktop --token="${TOKEN}"
```

Back in the browser, select Kubeconfig and select the “config” file from the folder C:\USERS\<username>\.kube, then click on Signin.

The dashboard should show up.

For setting up the dashboard on Ubuntu / Linux, follow instructions on this url: https://www.tutorialspoint.com/kubernetes/kubernetes_dashboard_setup.htm

Kubernetes Cluster Setup

Note – The setup is shown for Ubuntu machines. The same can be set up on other Linux machines as well.

Prerequisites

Installing Docker – Docker is required on all the instances of Kubernetes. Following are the steps to install the Docker.

Step 1 – Log on to the machine with the root user account.

Step 2 – Update the package information. Make sure that the apt package is working.

Step 3 – Run the following commands.

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

Step 4 – Add the new GPG key.

```
$ sudo apt-key adv \
   --keyserver hkp://ha.pool.sks-keyservers.net:80 \
   --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
$ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty
main" | sudo tee
/etc/apt/sources.list.d/docker.list
```

Step 5 – Update the API package image.

```
$ sudo apt-get update
```

Once all the above tasks are complete, you can start with the actual installation of the Docker engine. However, before this you need to verify that the kernel version you are using is correct.

Install Docker Engine

Run the following commands to install the Docker engine.

Step 1 – Logon to the machine.

Step 2 – Update the package index.

```
$ sudo apt-get update
```

Step 3 – Install the Docker Engine using the following command.

```
$ sudo apt-get install docker-engine
```

Step 4 – Start the Docker daemon.

```
$ sudo apt-get install docker-engine
```

Step 5 – To very if the Docker is installed, use the following command.

```
$ sudo docker run hello-world
```

Install etcd 2.0

This needs to be installed on Kubernetes Master Machine. In order to install it, run the following commands.

```
$ curl -L
https://github.com/coreos/etcd/releases/download/v2.0.0/etcd
-v2.0.0-linux-amd64.tar.gz -o etcd-v2.0.0-linux-amd64.tar.gz ->1
$ tar xzvf etcd-v2.0.0-linux-amd64.tar.gz ------>2
$ cd etcd-v2.0.0-linux-amd64 ------------>3
$ mkdir /opt/bin ------------->4
$ cp etcd* /opt/bin ------------>5
```

In the above set of command –

- First, we download the **etcd**. Save this with specified name.
- Then, we have to un-tar the tar package.
- We make a dir. inside the /opt named bin.
- Copy the extracted file to the target location.

Now we are ready to build Kubernetes. We need to install Kubernetes on all the machines on the cluster.

```
$ git clone https://github.com/GoogleCloudPlatform/kubernetes.git
$ cd kubernetes
$ make release
```

The above command will create a **_output** dir in the root of the kubernetes folder. Next, we can extract the directory into any of the directory of our choice /opt/bin, etc.

Next, comes the networking part wherein we need to actually start with the setup of Kubernetes master and node. In order to do this, we will make an entry in the host file which can be done on the node machine.

```
$ echo "<IP address of master machine> kube-master
< IP address of Node Machine>" >> /etc/hosts
```

Following will be the output of the above command.

```
root@boot2docker:/etc# cat /etc/hosts
127.0.0.1 boot2docker localhost localhost.local

# The following lines are desirable for IPv6 capable hosts
# (added automatically by netbase upgrade)

::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts

10.11.50.12 kube-master
10.11.50.11  kube-minion
```

Now, we will start with the actual configuration on Kubernetes Master.

First, we will start copying all the configuration files to their correct location.

```
$ cp <Current dir. location>/kube-apiserver /opt/bin/
$ cp <Current dir. location>/kube-controller-manager /opt/bin/
$ cp <Current dir. location>/kube-kube-scheduler /opt/bin/
$ cp <Current dir. location>/kubecfg /opt/bin/
$ cp <Current dir. location>/kubectl /opt/bin/
$ cp <Current dir. location>/kubernetes /opt/bin/
```

The above command will copy all the configuration files to the required location. Now we will come back to the same directory where we have built the Kubernetes folder.

```
$ cp kubernetes/cluster/ubuntu/init_conf/kube-apiserver.conf
/etc/init/
$ cp kubernetes/cluster/ubuntu/init_conf/kube-controller-
manager.conf /etc/init/
$ cp kubernetes/cluster/ubuntu/init_conf/kube-kube-scheduler.conf
/etc/init/

$ cp kubernetes/cluster/ubuntu/initd_scripts/kube-apiserver
/etc/init.d/
$ cp kubernetes/cluster/ubuntu/initd_scripts/kube-controller-
manager /etc/init.d/
$ cp kubernetes/cluster/ubuntu/initd_scripts/kube-kube-scheduler
/etc/init.d/

$ cp kubernetes/cluster/ubuntu/default_scripts/kubelet
/etc/default/
$ cp kubernetes/cluster/ubuntu/default_scripts/kube-proxy
/etc/default/
$ cp kubernetes/cluster/ubuntu/default_scripts/kubelet
/etc/default/
```

The next step is to update the copied configuration file under /etc. dir.

Configure etcd on master using the following command.

```
$ ETCD_OPTS = "-listen-client-urls = http://kube-master:4001"
```

Configure kube-apiserver

For this on the master, we need to edit the **/etc/default/kube-apiserver** file which we copied earlier.

```
$ KUBE_APISERVER_OPTS = "--address = 0.0.0.0 \
--port = 8080 \
--etcd_servers = <The path that is configured in ETCD_OPTS> \
--portal_net = 11.1.1.0/24 \
--allow_privileged = false \
--kubelet_port = < Port you want to configure> \
--v = 0"
```

Configure the kube Controller Manager

We need to add the following content in **/etc/default/kube-controller-manager**.

```
$ KUBE_CONTROLLER_MANAGER_OPTS = "--address = 0.0.0.0 \
```

```
--master = 127.0.0.1:8080 \
--machines = kube-minion \ -----> #this is the kubernatics node
--v = 0
```

Next, configure the kube scheduler in the corresponding file.

```
$ KUBE_SCHEDULER_OPTS = "--address = 0.0.0.0 \
--master = 127.0.0.1:8080 \
--v = 0"
```

Once all the above tasks are complete, we are good to go ahead by bring up the Kubernetes Master. In order to do this, we will restart the Docker.

```
$ service docker restart
```

Kubernetes Node Configuration

Kubernetes node will run two services the **kubelet and the kube-proxy**. Before moving ahead, we need to copy the binaries we downloaded to their required folders where we want to configure the kubernetes node.

Use the same method of copying the files that we did for kubernetes master. As it will only run the kubelet and the kube-proxy, we will configure them.

```
$ cp <Path of the extracted file>/kubelet /opt/bin/
$ cp <Path of the extracted file>/kube-proxy /opt/bin/
$ cp <Path of the extracted file>/kubecfg /opt/bin/
$ cp <Path of the extracted file>/kubectl /opt/bin/
$ cp <Path of the extracted file>/kubernetes /opt/bin/
```

Now, we will copy the content to the appropriate dir.

```
$ cp kubernetes/cluster/ubuntu/init_conf/kubelet.conf /etc/init/
$ cp kubernetes/cluster/ubuntu/init_conf/kube-proxy.conf
/etc/init/
$ cp kubernetes/cluster/ubuntu/initd_scripts/kubelet /etc/init.d/
$ cp kubernetes/cluster/ubuntu/initd_scripts/kube-proxy
/etc/init.d/
$ cp kubernetes/cluster/ubuntu/default_scripts/kubelet
/etc/default/
$ cp kubernetes/cluster/ubuntu/default_scripts/kube-proxy
/etc/default/
```

We will configure the **kubelet** and **kube-proxy conf** files.

We will configure the **/etc/init/kubelet.conf**.

```
$ KUBELET_OPTS = "--address = 0.0.0.0 \
--port = 10250 \
--hostname_override = kube-minion \
--etcd_servers = http://kube-master:4001 \
--enable_server = true
--v = 0"
/
```

For kube-proxy, we will configure using the following command.

```
$ KUBE_PROXY_OPTS = "--etcd_servers = http://kube-master:4001 \
--v = 0"
/etc/init/kube-proxy.conf
```

Finally, we will restart the Docker service.

```
$ service docker restart
```

Now we are done with the configuration. You can check by running the following commands.

```
$ /opt/bin/kubectl get minions
```

Tutorial #1: Quick Start

Start minikube

```
minikube start
```

Run the hello-minikube app

```
kubectl run hello-minikube --image=k8s.gcr.io/echoserver:1.10 --
port=8080
```

Create a service to expose hello-minikube Pod as a Kubernetes service

```
kubectl expose deployment hello-minikube --type=NodePort
```

```
# 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:
# can use "kubectl get pods" as well
$ kubectl get pod
```

View the service

```
kubectl get services
```

```
# We can see that the pod is now Running and we will now be able to
curl it:
$ curl $(minikube service hello-minikube --url)
```

Delete the service

```
kubectl delete services hello-minikube
```

Delete the deployment app

```
kubectl delete deployment hello-minikube
```

Stop minikube

```
minikube stop
```

OR

```
minikube ssh (docker / tcuser)
```

```
sudo poweroff
```

Tutorial #2

Create a cluster

Check that it is properly installed, by running the *minikube version* command:

```
minikube version
```

Start the cluster (*if not already done*), by running the *minikube start* command:

```
minikube start
```

To check if kubectl is installed you can run the *kubectl version* command:

```
kubectl version
```

Let's view the cluster details. We'll do that by running *kubectl cluster-info*:

```
kubectl cluster-info
```

To view the nodes in the cluster, run the *kubectl get nodes* command:

```
kubectl get nodes
```

Deploy an app

Check that kubectl is configured to talk to your cluster, by running the kubectl version command:

```
kubectl version
```

To view the nodes in the cluster, run the kubectl get nodescommand:

```
kubectl get nodes
```

Let's run our first app on Kubernetes with the kubectl runcommand. The run command creates a new deployment. We need to provide the deployment name and app image location (include the full repository url for images hosted outside Docker hub). We want to run the app on a specific port so we add the --port parameter:

```
kubectl run kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1 --port=8080
```

Great! You just deployed your first application by creating a deployment. This performed a few things for you:

- searched for a suitable node where an instance of the application could be run (we have only 1 available node)
- scheduled the application to run on that Node

- configured the cluster to reschedule the instance on a new Node when needed

To list your deployments use the get deployments command:

```
kubectl get deployments
```

We see that there is 1 deployment running a single instance of your app. The instance is running inside a Docker container on your node.

View our app

Pods that are running inside Kubernetes are running on a private, isolated network. By default they are visible from other pods and services within the same kubernetes cluster, but not outside that network. When we use kubectl, we're interacting through an API endpoint to communicate with our application.

We will cover other options on how to expose your application outside the kubernetes cluster in Module 4.

The kubectl command can create a proxy that will forward communications into the cluster-wide, private network. The proxy can be terminated by pressing control-C and won't show any output while its running.

We will open a second terminal window to run the proxy.

```
kubectl proxy
```

We now have a connection between our host (the online terminal) and the Kubernetes cluster. The proxy enables direct access to the API from these terminals.

You can see all those APIs hosted through the proxy endpoint, now available at through http://localhost:8001. For example, we can query the version directly through the API using the curl command:

```
curl http://localhost:8001/version
```

The API server will automatically create an endpoint for each pod, based on the pod name, that is also accessible through the proxy.

First we need to get the Pod name, and we'll store in the environment variable POD_NAME:

```
export POD_NAME=$(kubectl get pods -o go-template --template '{{range
.items}}{{.metadata.name}}{{"\n"}}{{end}}')
```

```
echo Name of the Pod: $POD_NAME
```

Now we can make an HTTP request to the application running in that pod:

```
curl http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME/proxy/
```

The url is the route to the API of the Pod.

Viewing Pods and Nodes

Let's verify that the application we deployed in the previous scenario is running. We'll use the kubectl get command and look for existing Pods:

```
kubectl get pods
```

If no pods are running, list the Pods again.

Next, to view what containers are inside that Pod and what images are used to build those containers we run the describe pods command:

```
kubectl describe pods
```

We see here details about the Pod's container: IP address, the ports used and a list of events related to the lifecycle of the Pod.

Show app in the terminal

(Can be taken from previous module)

Run the proxy:

```
kubectl proxy
```

Now again, we'll get the Pod name and query that pod directly through the proxy. To get the Pod name and store it in the POD_NAME environment variable:

```
export POD_NAME=$(kubectl get pods -o go-template --template '{{range
.items}}{{.metadata.name}}{{"\n"}}{{end}}') echo Name of the Pod: $POD_NAME
```

To see the output of our application, run a curl request.

```
curl http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME/proxy/
```

The url is the route to the API of the Pod.

View Container logs

Anything that the application would normally send to STDOUT becomes logs for the container within the Pod. We can retrieve these logs using the kubectl logs command:

```
kubectl logs $POD_NAME
```

Executing command on the container

We can execute commands directly on the container once the Pod is up and running. For this, we use the exec command and use the name of the Pod as a parameter. Let's list the environment variables:

```
kubectl exec $POD_NAME env
```

Again, worth mentioning that the name of the container itself can be omitted since we only have a single container in the Pod.

Next let's start a bash session in the Pod's container:

```
kubectl exec -ti $POD_NAME bash
```

We have now an open console on the container where we run our NodeJS application. The source code of the app is in the server.js file:

```
cat server.js
```

You can check that the application is up by running a curl command:

```
curl localhost:8080
```

Note: here we used localhost because we executed the command inside the NodeJS container

To close your container connection type exit.

Using a Service to expose your app

Create a new service

Let's verify that our application is running. We'll use the kubectl get command and look for existing Pods:

```
kubectl get pods
```

Next let's list the current Services from our cluster:

```
kubectl get services
```

We have a Service called kubernetes that is created by default when minikube starts the cluster. To create a new service and expose it to external traffic we'll use the expose command with NodePort as parameter (minikube does not support the LoadBalancer option yet)

```
kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080
```

Let's run again the get services command:

```
kubectl get services
```

We have now a running Service called kubernetes-bootcamp. Here we see that the Service received a unique cluster-IP, an internal port and an external-IP (the IP of the Node).

To find out what port was opened externally (by the NodePort option) we'll run the describe service command:

```
kubectl describe services/kubernetes-bootcamp
```

Create an environment variable called NODE_PORT that has the value of the Node port assigned:

```
export NODE_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index
.spec.ports 0).nodePort}}')
```

```
echo NODE_PORT=$NODE_PORT
```

Now we can test that the app is exposed outside of the cluster using curl, the IP of the Node and the externally exposed port:

```
curl $(minikube ip):$NODE_PORT
```

And we get a response from the server. The Service is exposed.

Determine the IP address ad NODE_PORT values and try the url from a browser.

Using Labels

The Deployment created automatically a label for our Pod. With describe deployment command you can see the name of the label:

```
kubectl describe deployment
```

Let's use this label to query our list of Pods. We'll use the kubectl get pods command with -l as a parameter, followed by the label values:

```
kubectl get pods -l run=kubernetes-bootcamp
```

You can do the same to list the existing services:

```
kubectl get services -l run=kubernetes-bootcamp
```

Get the name of the Pod and store it in the POD_NAME environment variable:

```
export POD_NAME=$(kubectl get pods -o go-template --template '{{range
.items}}{{.metadata.name}}{{"\n"}}{{end}}')
```

```
echo Name of the Pod: $POD_NAME
```

To apply a new label we use the label command followed by the object type, object name and the new label:

```
kubectl label pod $POD_NAME app=v1
```

This will apply a new label to our Pod (we pinned the application version to the Pod), and we can check it with the describe pod command:

```
kubectl describe pods $POD_NAME
```

We see here that the label is attached now to our Pod. And we can query now the list of pods using the new label:

```
kubectl get pods -l app=v1
```

And we see the Pod.

Deleting a service

To delete Services you can use the `delete service`command. Labels can be used also here:

```
kubectl delete service -l run=kubernetes-bootcamp
```

Confirm that the service is gone:

```
kubectl get services
```

This confirms that our Service was removed. To confirm that route is not exposed anymore you can `curl` the previously exposed IP and port:

```
curl $(minikube ip):$NODE_PORT
```

This proves that the app is not reachable anymore from outside of the cluster. You can confirm that the app is still running with a curl inside the pod:

```
kubectl exec -ti $POD_NAME curl localhost:8080
```

We see here that the application is up.

Running multiple instances of your app

To list your deployments use the `get deployments` command: `kubectl get deployments`

We should have 1 Pod. If not, run the command again. This shows:

The DESIRED state is showing the configured number of replicas

The CURRENT state show how many replicas are running now

The UP-TO-DATE is the number of replicas that were updated to match the desired (configured) state

The AVAILABLE state shows how many replicas are actually AVAILABLE to the users

Next, let's scale the Deployment to 4 replicas. We'll use the kubectl scale command, followed by the deployment type, name and desired number of instances:

```
kubectl scale deployments/kubernetes-bootcamp --replicas=4
```

To list your Deployments once again, use get deployments:

```
kubectl get deployments
```

The change was applied, and we have 4 instances of the application available. Next, let's check if the number of Pods changed:

```
kubectl get pods -o wide
```

There are 4 Pods now, with different IP addresses. The change was registered in the Deployment events log. To check that, use the describe command:

```
kubectl describe deployments/kubernetes-bootcamp
```

You can also view in the output of this command that there are 4 replicas now.

Load balancing

Let's check that the Service is load-balancing the traffic. To find out the exposed IP and Port we can use the describe service as we learned in the previous Module:

Check if the service exists:

```
kubectl get services
```

```
kubectl describe services/kubernetes-bootcamp
```

Create the service if it doesn't exist:

```
kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080
```

```
kubectl describe services/kubernetes-bootcamp
```

Create an environment variable called NODE_PORT that has a value as the Node port:

```
export NODE_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index
.spec.ports 0).nodePort}}')
```

```
echo NODE_PORT=$NODE_PORT
```

Next, we'll do a curl to the exposed IP and port. Execute the command multiple times:

```
curl $(minikube ip):$NODE_PORT
```

We hit a different Pod with every request. This demonstrates that the load-balancing is working.

Scale down

To scale down the Service to 2 replicas, run again the scale command:

```
kubectl scale deployments/kubernetes-bootcamp --replicas=2
```

List the Deployments to check if the change was applied with the get deployments command:

```
kubectl get deployments
```

The number of replicas decreased to 2. List the number of Pods, with get pods:

```
kubectl get pods -o wide
```

This confirms that 2 Pods were terminated.

Performing a rolling update

Update the version of the app

To list your deployments use the get deployments command: `kubectl get deployments`

To list the running Pods use the get pods command:

```
kubectl get pods
```

To view the current image version of the app, run a describe command against the Pods (look at the Image field):

```
kubectl describe pods
```

To update the image of the application to version 2, use the set image command, followed by the deployment name and the new image version:

```
kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2
```

The command notified the Deployment to use a different image for your app and initiated a rolling update. Check the status of the new Pods, and view the old one terminating with the get pods command:

```
kubectl get pods
```

Verify an update

First, let's check that the App is running. To find out the exposed IP and Port we can use describe service:

```
kubectl describe services/kubernetes-bootcamp
```

Create an environment variable called NODE_PORT that has the value of the Node port assigned:

```
export NODE_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')
```

```
echo NODE_PORT=$NODE_PORT
```

Next, we'll do a `curl` to the the exposed IP and port:

```
curl $(minikube ip):$NODE_PORT
```

We hit a different Pod with every request and we see that all Pods are running the latest version (v2).

The update can be confirmed also by running a rollout status command:

```
kubectl rollout status deployments/kubernetes-bootcamp
```

To view the current image version of the app, run a describe command against the Pods:

```
kubectl describe pods
```

We run now version 2 of the app (look at the Image field)

Rollback an update

Let's perform another update, and deploy image tagged as `v10` :

```
kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=gcr.io/google-samples/kubernetes-bootcamp:v10
```

Use `get deployments` to see the status of the deployment:

```
kubectl get deployments
```

And something is wrong… We do not have the desired number of Pods available. List the Pods again:

```
kubectl get pods
```

A `describe` command on the Pods should give more insights:

```
kubectl describe pods
```

There is no image called `v10` in the repository. Let's roll back to our previously working version. We'll use the `rollout` undo command:

```
kubectl rollout undo deployments/kubernetes-bootcamp
```

The `rollout` command reverted the deployment to the previous known state (v2). Updates are versioned and you can revert to any previously known state of a Deployment. List again the Pods:

kubectl get pods

Four Pods are running. Check again the image deployed on the them: kubectl describe pods

We see that the deployment is using a stable version of the app (v2). The Rollback was successful.

Cleanup

Delete the service

kubectl get services

kubectl delete services kubernetes-bootcamp

kubectl get services

Delete the deployment

kubectl get deployments

kubectl delete deployment kubernetes-bootcamp

kubectl get deployments

Delete the pods

kubectl get pods –o wide

kubectl delete pods --all (or one-by-one)

kubectl get pods –o wide

Use a Service to Access an Application in a Cluster

https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/

Creating a service for an application running in two pods:

hello-application.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  selector:
    matchLabels:
      run: load-balancer-example
  replicas: 2
  template:
    metadata:
      labels:
        run: load-balancer-example
```

```
    spec:
      containers:
        - name: hello-world
          image: gcr.io/google-samples/node-hello:1.0
          ports:
            - containerPort: 8080
              protocol: TCP
```

1. Run a Hello World application in your cluster: Create the application Deployment using the file above:

```
kubectl apply -f https://k8s.io/examples/service/access/hello-application.yaml
```

The preceding command creates a Deployment object and an associated ReplicaSet object. The ReplicaSet has two Pods, each of which runs the Hello World application.

2. Display information about the Deployment:

```
kubectl get deployments hello-world

kubectl describe deployments hello-world
```

3. Display information about your ReplicaSet objects:

```
kubectl get replicasets

kubectl describe replicasets
```

4. Create a Service object that exposes the deployment:

```
kubectl expose deployment hello-world --type=NodePort --name=example-service
```

5. Display information about the Service:

```
kubectl describe services example-service
```

The output is similar to this:

```
Name:                   example-service
Namespace:              default
Labels:                 run=load-balancer-example
Annotations:            <none>
Selector:               run=load-balancer-example
Type:                   NodePort
IP:                     10.32.0.16
Port:                   <unset> 8080/TCP
TargetPort:             8080/TCP
NodePort:               <unset> 31496/TCP
Endpoints:              10.200.1.4:8080,10.200.2.5:8080
Session Affinity:       None
Events:                 <none>
```

Make a note of the NodePort value for the service. For example, in the preceding output, the NodePort value is 31496.

6. List the pods that are running the Hello World application:

```
kubectl get pods --selector="run=load-balancer-example" --output=wide
```

The output is similar to this:

```
NAME                                  READY    STATUS     ...  IP            NODE

hello-world-2895499144-bsbk5   1/1      Running    ...  10.200.1.4    worker1

hello-world-2895499144-m1pwt   1/1      Running    ...  10.200.2.5    worker2
```

7. Get the public IP address of one of your nodes that is running a Hello World pod. How you get this address depends on how you set up your cluster. For example, if you are using Minikube, you can see the node address by running `kubectl cluster-info`. If you are using Google Compute Engine instances, you can use the `gcloud compute instances list` command to see the public addresses of your nodes.
8. On your chosen node, create a firewall rule that allows TCP traffic on your node port. For example, if your Service has a NodePort value of 31568, create a firewall rule that allows TCP traffic on port 31568. Different cloud providers offer different ways of configuring firewall rules.
9. Use the node address and node port to access the Hello World application:

```
curl http://<public-node-ip>:<node-port>
```

where `<public-node-ip>` is the public IP address of your node, and `<node-port>` is the NodePort value for your service. The response to a successful request is a hello message:

```
Hello Kubernetes!
```

Kubernetes Secrets

Kubernetes `secret` objects let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a `secret` is safer and more flexible than putting it verbatim in a Pod definition or in a container image.

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure.

Users can create secrets, and the system also creates some secrets.

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in two ways: as files in a volume mounted on one or more of its containers, or used by kubelet when pulling images for the pod.

Built-in Secrets

Service Accounts Automatically Create and Attach Secrets with API Credentials

Kubernetes automatically creates secrets which contain credentials for accessing the API and it automatically modifies your pods to use this type of secret.

The automatic creation and use of API credentials can be disabled or overridden if desired. However, if all you need to do is securely access the apiserver, this is the recommended workflow.

See the Service Account documentation for more information on how Service Accounts work.

Creating your own Secrets

Creating a Secret Using kubectl create secret

Say that some pods need to access a database. The username and password that the pods should use is in the files `./username.txt` and `./password.txt` on your local machine.

```
# Create files needed for rest of example.
echo -n 'admin' > ./username.txt
echo -n '1f2d1e2e67df' > ./password.txt
```

The `kubectl create secret` command packages these files into a Secret and creates the object on the Apiserver.

```
kubectl create secret generic db-user-pass --from-file=./username.txt -
-from-file=./password.txt
secret "db-user-pass" created
```

Note: Special characters such as `$`, `\*`, and `!` require escaping. If the password you are using has special characters, you need to escape them using the `\\` character. For example, if your actual password is `S!B\*d$zDsb`, you should execute the command this way: `kubectl create`

`secret generic dev-db-secret --from-literal=username=devuser --from-literal=password=S\!B\\*d\$zDsb`. You do not need to escape special characters in passwords from files (`--from-file`).

You can check that the secret was created like this:

```
kubectl get secrets
NAME                  TYPE                                  DATA      AGE
db-user-pass          Opaque                                2         51s
kubectl describe secrets/db-user-pass
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password.txt:    12 bytes
username.txt:    5 bytes
```

Note: `kubectl get` and `kubectl describe` avoid showing the contents of a secret by default. This is to protect the secret from being exposed accidentally to an onlooker, or from being stored in a terminal log.

Creating a Secret Manually

You can also create a Secret in a file first, in json or yaml format, and then create that object. The Secret contains two maps: data and stringData. The data field is used to store arbitrary data, encoded using base64. The stringData field is provided for convenience, and allows you to provide secret data as unencoded strings.

For example, to store two strings in a Secret using the data field, convert them to base64 as follows:

```
echo -n 'admin' | base64
YWRtaW4=
echo -n '1f2d1e2e67df' | base64
MWYyZDFlMmU2N2Rm
```

To decode:

```
echo "MWYyZDFlMmU2N2Rm" | base64 --decode
```

Write a Secret that looks like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
```

```
  password: MWYyZDFlMmU2N2Rm
```

Now create the Secret using `kubectl apply`:

```
kubectl apply -f ./secret.yaml
secret "mysecret" created
```

Read secret data:

```
kubectl get secret mysecret -o jsonpath="{.data.username}" | base64 --decode
```

For certain scenarios, you may wish to use the stringData field instead. This field allows you to put a non-base64 encoded string directly into the Secret, and the string will be encoded for you when the Secret is created or updated.

A practical example of this might be where you are deploying an application that uses a Secret to store a configuration file, and you want to populate parts of that configuration file during your deployment process.

If your application uses the following configuration file:

```
apiUrl: "https://my.api.com/api/v1"
username: "user"
password: "password"
```

You could store this in a Secret using the following:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  config.yaml: |-
    apiUrl: "https://my.api.com/api/v1"
    username: {{username}}
    password: {{password}}
```

Your deployment tool could then replace the `{{username}}` and `{{password}}` template variables before running `kubectl apply`.

stringData is a write-only convenience field. It is never output when retrieving Secrets. For example, if you run the following command:

```
kubectl get secret mysecret -o yaml
```

The output will be similar to:

```
apiVersion: v1
kind: Secret
metadata:
```

```
  creationTimestamp: 2018-11-15T20:40:59Z
  name: mysecret
  namespace: default
  resourceVersion: "7225"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: c280ad2e-e916-11e8-98f2-025000000001
type: Opaque
data:
  config.yaml:
YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXNlcm5hbWU6IHt7dXNlcm5hbWV
9fQpwYXNzd29yZDoge3twYXNzd29yZH19
```

If a field is specified in both data and stringData, the value from stringData is used. For example, the following Secret definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
stringData:
  username: administrator
```

Results in the following secret:

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:46:46Z
  name: mysecret
  namespace: default
  resourceVersion: "7579"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 91460ecb-e917-11e8-98f2-025000000001
type: Opaque
data:
  username: YWRtaW5pc3RyYXRvcg==
```

Where `YWRtaW5pc3RyYXRvcg==` decodes to `administrator`.

The keys of data and stringData must consist of alphanumeric characters, '-', '_' or '.'.

Encoding Note: The serialized JSON and YAML values of secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted. When using the `base64` utility on Darwin/macOS users should avoid using the `-b` option to split long lines. Conversely Linux users *should* add the option `-w 0` to `base64` commands or the pipeline `base64 | tr -d '\n'` if `-w` option is not available.

Using Secrets

- https://v1-13.docs.kubernetes.io/docs/concepts/configuration/secret/
- https://www.tutorialspoint.com/kubernetes/kubernetes_secrets.htm

Using Secrets with Volumes

UsingSecretVolume.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: gcr.io/google-samples/node-hello:1.0
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

```
kubectl apply -f ./UsingSecretVolume.yaml
```

Consuming Secret Values from Volumes

Inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base-64 decoded and stored inside these files.

To connect to the container (pod):

```
kubectl exec -it mypod sh
```

This is the result of commands executed inside the container from the example above:

```
ls /etc/foo/
username
password
cat /etc/foo/username
admin
cat /etc/foo/password
1f2d1e2e67df
```

Using Secrets as Environment Variables

UsingSecretEnv.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: gcr.io/google-samples/node-hello:1.0
```

```
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

```
kubectl apply -f ./UsingSecretEnv.yaml
```

Consuming Secret Values from Volumes

Inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base-64 decoded and stored inside these files.

To connect to the container (pod):

```
kubectl exec -it secret-env-pod sh
```

This is the result of commands executed inside the container from the example above:

```
echo $SECRET_USERNAME
admin
echo $SECRET_PASSWORD
1f2d1e2e67df
```

Cleanup

```
kubectl delete delpoyments/secret-env-pod
```

```
kubectl delete delpoyments/mypod
```

Create a Kubernetes dev space with Azure Dev Spaces (.NET Core and VS Code)

Pre-reqs

- Azure subscription
- VS Code
- Azure CLI ver 2.0.43 or higher
- Kubernetes cluster running Kubernetes 1.9.6 or later, in the EastUS, EastUS2, CentralUS, WestUS2, WestEurope, SoutheastAsia, CanadaCentral, or CanadaEast region, with **Http Application Routing** enabled
- **Steps**:
  - Create RG

```
az group create --name MyResourceGroup --location eastus
```

  - Create AKS

```
az aks create -g MyResourceGroup -n MyAKS --location eastus --kubernetes-version 1.10.9 --enable-addons http_application_routing
```

Setup Azure DevSpaces

  - Set up DevSpaces on your AKS cluster

```
az aks use-dev-spaces -g MyResourceGroup -n MyAKS
```

  - Download the Azure Dev Spaces extension for VS Code (https://marketplace.visualstudio.com/items?itemName=azuredevspaces.azds)
  - Click Install once on the extension's Marketplace page, and again in VS Code
  - Download the Azure Dev Spaces CLI from https://aka.ms/get-azds-windows

Build and run the code

  - Download sample code from GitHub: https://github.com/Azure/dev-spaces
  - Change directory to the webfrontend folder:

```
cd dev-spaces/samples/dotnetcore/getting-started/webfrontend
```

  - Generate Docker and Helm chart assets:

```
azds prep --public
```

  - Select the default Azure Dev Space

```
azds space select
```

- Enter “1” to select the default Dev Space to continue and then enter “y” to continue
- Build and run your code in AKS. In the terminal window from the webfrontend folder, run this command:

```
azds up
```

- Scan the console output for information about the URL that was created by the up command. It will be in the form:

```
(pending registration) Service 'webfrontend' port 'http' will be available at
<url>\r\nService 'webfrontend' port 80 (TCP) is available at
http://localhost:<port>
```

Open this URL in a browser window, and you should see the web app load.

Update a content file

1. Locate a file, such as `./Views/Home/Index.cshtml`, and make an edit to the HTML. For example, change line 7 that reads `<h2>Application uses</h2>` to something like: `<h2>Hello k8s in Azure!</h2>`
2. Save the file. Moments later, in the Terminal window you'll see a message saying a file in the running container was updated.
3. Go to your browser and refresh the page. You should see the web page display the updated HTML.

What happened? Edits to content files, like HTML and CSS, don't require recompilation in a .NET Core web app, so an active `azds up` command automatically syncs any modified content files into the running container in Azure, so you can see your content edits right away.

Update a code file

Updating code files requires a little more work, because a .NET Core app needs to rebuild and produce updated application binaries.

1. In the terminal window, press `Ctrl+C` (to stop `azds up`).
2. Open the code file named `Controllers/HomeController.cs`, and edit the message that the About page will display: `ViewData["Message"] = "Your application description page.";`
3. Save the file.
4. Run `azds up` in the terminal window.

This command rebuilds the container image and redeploys the Helm chart. To see your code changes take effect in the running application, go to the About menu in the web app.

Debug a container in Kubernetes

In this section, you'll use VS Code to directly debug your container running in Azure. You'll also learn how to get a faster edit-run-test loop.

Initialize debug assets with the VS Code extension

You first need to configure your code project so VS Code will communicate with the dev space in Azure. The VS Code extension for Azure Dev Spaces provides a helper command to set up debug configuration.

Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command:

```
Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces
```

This adds debug configuration for Azure Dev Spaces under the .vscode folder. This command is not to be confused with the azds prep command, which configures the project for deployment.

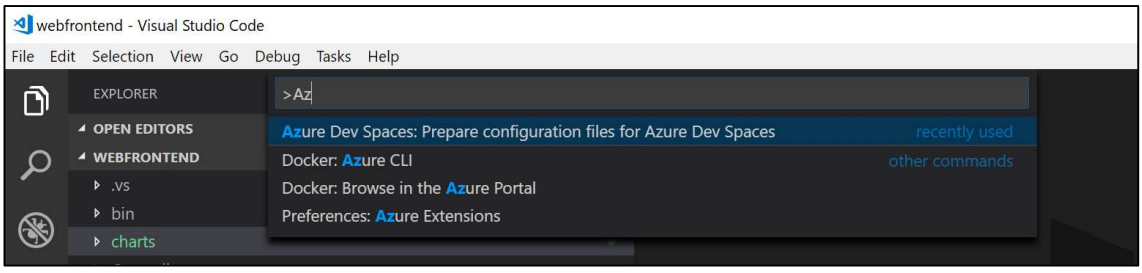
Initialize debug assets with the VS Code extension

You first need to configure your code project so VS Code will communicate with the dev space in Azure. The VS Code extension for Azure Dev Spaces provides a helper command to set up debug configuration.

Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command:

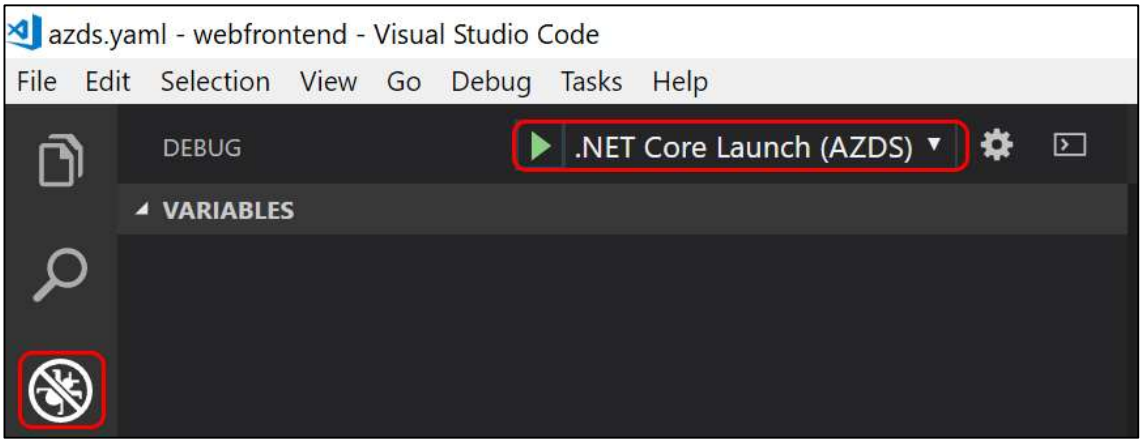
```
Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces
```

This adds debug configuration for Azure Dev Spaces under the .vscode folder. This command is not to be confused with the azds prep command, which configures the project for deployment.



Select the AZDS debug configuration

1. To open the Debug view, click on the Debug icon in the **Activity Bar** on the side of VS Code.
2. Select **.NET Core Launch (AZDS)** as the active debug configuration.



Debug the container in Kubernetes

Hit **F5** to debug your code in Kubernetes.

As with the `up` command, code is synced to the dev space, and a container is built and deployed to Kubernetes. This time, of course, the debugger is attached to the remote container.

The VS Code status bar will display a clickable URL.

Set a breakpoint in a server-side code file, for example within the `Index()` function in the `Controllers/HomeController.cs` source file. Refreshing the browser page causes the breakpoint to be hit.

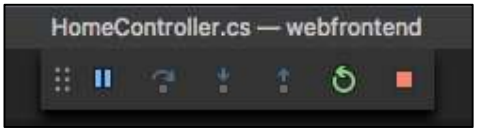
You have full access to debug information just like you would if the code was executing locally, such as the call stack, local variables, exception information, etc.

Edit code and refresh

With the debugger active, make a code edit. For example, modify the About page's message in `Controllers/HomeController.cs`.

```
public IActionResult About()
{
    ViewData["Message"] = "My custom message in the About page.";
    return View();
}
```

Save the file, and in the **Debug actions pane**, click the **Refresh** button.



Instead of rebuilding and redeploying a new container image each time code edits are made, which will often take considerable time, Azure Dev Spaces will incrementally recompile code within the existing container to provide a faster edit/debug loop.

Refresh the web app in the browser, and go to the About page. You should see your custom message appear in the UI.

Join a new node to the master

```
sudo kubeadm token create --print-join-command

kubeadm join 172.20.37.25:6443 --token <token> --discovery-token-ca-cert-hash sha2
56:a968c697abcae8beafe51738ede08f2888003e01e1aa5cca1a7eb8b69ef79767
```

Expose Deployment as a Service

Example 1: Direct Run

```
kubectl run hello-world --replicas=5 --labels="run=load-balancer-
example" --image=gcr.io/google-samples/node-hello:1.0  --port=8080
```

```
kubectl get deployments hello-world
kubectl describe deployments hello-world
```

```
kubectl get replicasets
kubectl describe replicasets
```

```
kubectl expose deployment hello-world --type=NodePort --name=my-service
```

```
kubectl get services my-service
```

```
kubectl describe services my-service
```

Note the NodePort value

```
kubectl cluster-info
```

Note the ip address of the k8s cluster

```
curl http://<ip>:<port>
```

OR in browser:

```
http://< ip>:<port>
```

Example 2: With a YAML file

nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

```
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

```
kubectl apply -f nginx-deployment.yaml
```

OR

```
kubectl apply -f https://k8s.io/examples/controllers/nginx-
deployment.yaml
```

```
kubectl get deployments
```

```
kubectl get rs
```

```
kubectl expose deployment nginx-deployment --type=NodePort --
name=nginx-service
```

```
kubectl get services nginx-service
```

```
kubectl describe services nginx-service
```

Note the NodePort value

```
kubectl cluster-info
```

Note the ip address of the k8s cluster

```
curl http://<ip>:<port>
```

OR in browser:

```
http://< ip>:<port>
```