Concepts

Overview

01_What is Kubernetes

Containers

What are containers?

• Containers are completely isolated environments, as in they can have their own processes or services, their own network interfaces, their own mounts, just like Virtual machines, except that they all share the same OS kernel.
• Docker containers share the underlying kernel.
• Let us say we have a system with an Uburtu OS with Docker installed on it. Docker can run any flavor of OS on top of it if they are all based on the same kernel — in this case Linux docker utilizes the underlying kernel of the Docker host which works with all OS's above
• Some of the different types of containers are LXC, LXD, LXCFS etc. Docker utilizes LXC containers.
• Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight.
• Similar to a VM, a container has its own filesystem, share of CPU, memory, process space, and more.
• As they are decoupled from the underlying infrastructure, they are portable across clouds and various OS distributions.

So, what is an OS that does not share the same kernel as these?

• Windows
• You will not be able to run a windows-based container on a Docker host with Linux OS on it.

• Unlike hypervisors, Docker is not meant to virtualize and run different Operating systems and kernels on the same hardware. The main purpose of Docker is to containerize applications and to ship them and run them

• Docker has relaxed isolation as more resources are shared between containers like the kernel etc.

• Whereas VMs have complete isolation from each other. Since VMs do not rely on the underlying OS or kernel, you can run different types of OS such as linux based or windows based on the same hypervisor

Difference between the images and containers.

◇ An image is a package or a template, just like a VM template that you might have worked with in the virtualization world. It is used to create one or more containers.
◇ Containers are running instances of images that are isolated and have their own environments and set of processes.
◇ With Docker, a major portion of work involved in setting up the infrastructure is now in the hands of the developers in the form of a Docker file.

Container orchestration
◇ This whole process of automatically deploying and managing containers is known as Container Orchestration
◇ This orchestration platform needs to orchestrate the connectivity between the containers and automatically scale up or down based on the load.

Orchestration technologies
◇ Kubernetes is thus a container orchestration technology. There are multiple such technologies available today —

1. Docker Swarm from Docker.
2. Kubernetes from Google.
3. Mesos from Apache.

Containers have become popular because they provide extra benefits, such as:

1. Agile application creation and deployment:
2. Continuous development, integration, and deployment.
3. Environmental consistency across development, testing, and production
4. Resource isolation
5. Resource utilization
6. Cloud and OS distribution portability
7. Application-centric management
8. Loosely coupled, distributed, elastic, liberated micro-services
9. Dev and Ops separation of concerns:
10. Observability:

1. **Agile application creation and deployment**: increased ease and efficiency of container image creation compared to VM image use.
2. **Continuous development, integration, and deployment**: provides for reliable and frequent container image build and deployment with quick and efficient rollbacks (due to image immutability).
3. **Environmental consistency across development, testing, and production**: Runs the same on a laptop as it does in the cloud.
4. **Resource isolation**: predictable application performance.
5. **Resource utilization**: high efficiency and density.
6. **Cloud and OS distribution portability**: Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
7. **Application-centric management**: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
8. **Loosely coupled, distributed, elastic, liberated micro-services**: applications are broken into smaller, independent components and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
9. **Dev and Ops separation of concerns**: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
10. **Observability**: not only surfaces OS-level information and metrics, but also application health and other signals.

What is Kubernetes

What is Kubernetes?

• Kubernetes is a container Orchestration technology used to orchestrate the deployment and management of 100s and 1000s of containers in a clustered environment

• Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

Why do you need Kubernetes and what can it do?

◇ Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?
◇ That's how Kubernetes comes to the rescue!
◇ **Kubernetes provides you with a framework to run distributed systems resiliently.**
◇ It takes care of scaling and failover for your application, provides deployment patterns, and more.
◇ For example, Kubernetes can easily manage a canary deployment for your system.

Canary Deployment

In software engineering, canary deployment is the practice of making staged releases.
We roll out a software update to a small part of the users first, so they may test it and provide feedback.
Once the change is accepted, the update is rolled out to the rest of the users

Kubernetes provides you with:

1. Service discovery and load balancing:
2. Storage orchestration
3. Automated rollouts and rollbacks
4. Automatic bin packing
5. Self-healing
6. Secret and configuration management

1. **Service discovery and load balancing:** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
2. **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
3. **Automated rollouts and rollbacks:** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
4. **Automatic bin packing:** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
5. **Self-healing:** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
6. **Secret and configuration management:** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

Kubernetes Advantages
◇ Your application is now highly available as hardware failures do not bring your application down because you have multiple instances of your application running on different nodes.
◇ The user traffic is load balanced across the various containers.
◇ When demand increases, deploy more instances of the application seamlessly and within a matter of seconds and we can do that at a service level.
◇ When we run out of hardware resources, scale the number of nodes up/down without having to take down the application.
◇ And do all these easily with a set of declarative object configuration files.

What Kubernetes is not?
◇ Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system.
◇ Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and

alerting solutions.
◇ However, Kubernetes is not monolithic, and these default solutions are optional and pluggable.
◇ Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

Kubernetes:
1. Does not limit the types of applications supported.
2. Does not deploy source code and does not build your application
3. Does not provide application-level services
4. Does not dictate logging, monitoring, or alerting solutions
5. Does not provide nor mandate a configuration language/system (for example, Jsonnet)
6. Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems

1. **Does not limit the types of applications supported.** Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.

2. **Does not deploy source code and does not build your application.** Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.

3. **Does not provide application-level service**s: such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.

4. **Does not dictate logging, monitoring, or alerting solutions**. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.

5. **Does not provide nor mandate a configuration language/system (for example, Jsonnet)**. It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.

6. **Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.**

7. **Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration.** The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

02_Kubernetes_components

MASTER

Kubernetes Components

• Cluster

◇ When you deploy Kubernetes, you get a cluster.
◇ A cluster is a set of nodes grouped together.
◇ Cluster provides fault-tolerance
■ This way even if one node fails, your application is still accessible from the other nodes. (providing fault-tolerance)
◇ Moreover, having multiple nodes helps in sharing load as well. (providing high availability)

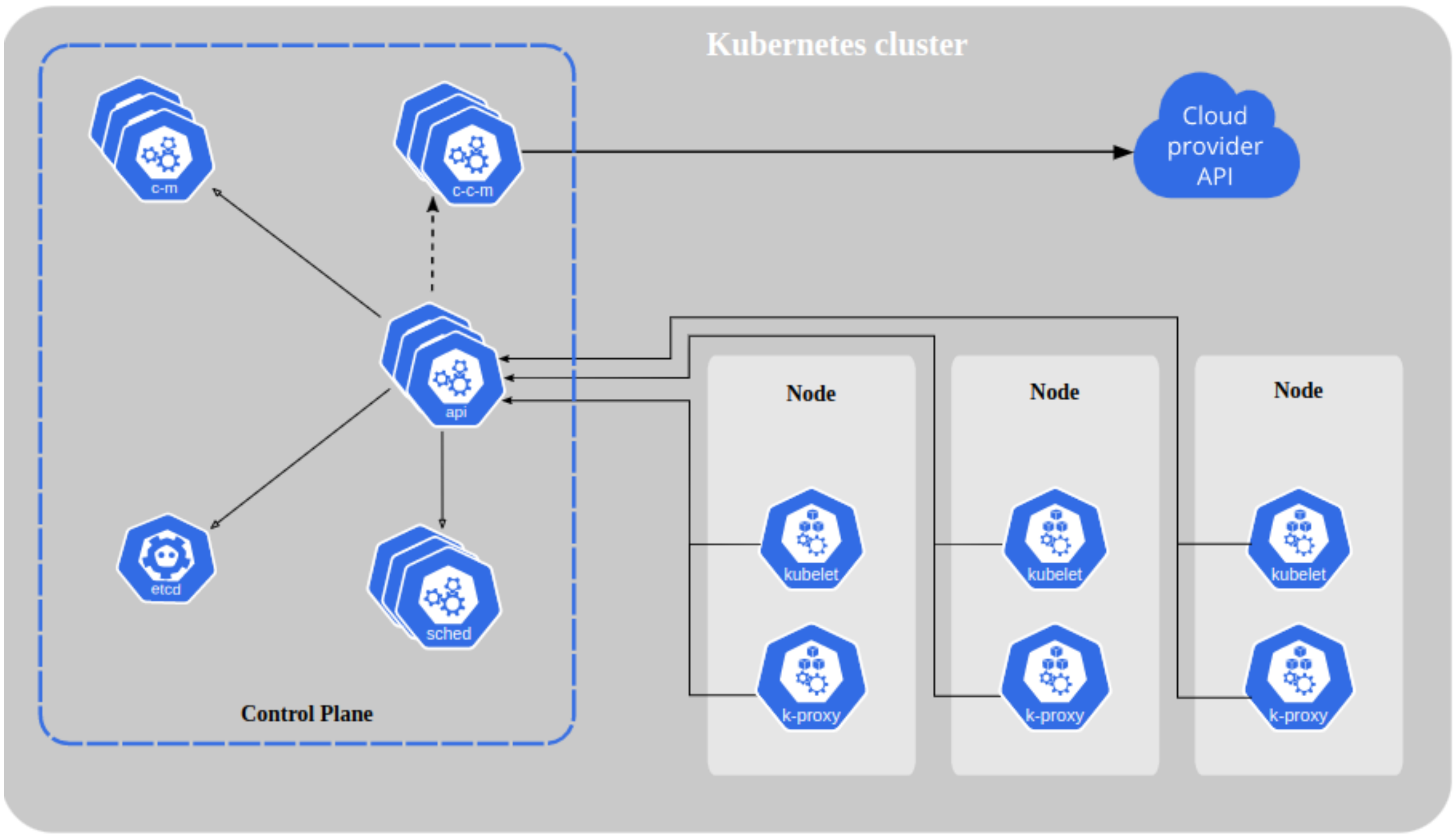
• Nodes

◇ A node is a physical or virtual machine — on which kubernetes is installed.
◇ A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications.
◇ Every cluster has at least one worker node.

• The worker node(s) host the Pods

◇ A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.

• The Control Plane
◇ The control plane manages the worker nodes and the Pods in the cluster.
◇ In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.



Control Plane Components

◇ The control plane's components make global decisions about the cluster, as well as detecting and responding to cluster events

■ (for example, scheduling),

■ (for example, starting up a new pod when a deployment's `replicas` field is unsatisfied).

◇ Control plane components can be run on any machine in the cluster.

◇ However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine.

■ See Creating Highly Available clusters with kubeadm for an example control plane setup that runs across multiple machines.

Control Plane Components

1. Kube apiserver (api)
2. etcd   (etcd)
3. Kube scheduler (sched)
4. Kube controller-manager (c-m)
5. Kube Cloud-controller-manager (c-c-m)

Kube-apiserver (api)

◇ The Kube API server is a component of the Kubernetes control plane that exposes the Kubernetes API.
◇ The Kube API server is the front end for the Kubernetes control plane.
◇ The main implementation of a Kubernetes API server is kube-apiserver.
◇ kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances.
◇ You can run several instances of kube-apiserver and balance traffic between those instances.
◇ The users, management devices, Command line interfaces all talk to the API server to interact with the kubernetes cluster

etcd   (etcd)

◇ ETCD is a distributed, reliable key-value store used by kubernetes to store all data used to manage the cluster.
◇ Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.
◇ If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for that data.
◇ You can find in-depth information about etcd in the official documentation.
◇ when you have multiple nodes and multiple masters in your cluster, etcd stores all that information on all the nodes in the cluster in a distributed manner
◇ ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters

Kube-scheduler (sched)

◇ Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

◇ It looks for newly created containers and assigns them to Nodes
◇ The scheduler is responsible for distributing work or containers across multiple nodes.
◇ Factors taken into account for scheduling decisions include:

1. individual and collective resource requirements,
2. hardware/software/policy constraints,
3. affinity and anti-affinity specifications,
4. data locality,
5. inter-workload interference, and
6. deadlines.

Kube-controller-manager (c-m)

◇ Control plane component that runs controller processes.
◇ Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some types of these controllers are:
1. **Node controller:** Responsible for noticing and responding when nodes go down.
2. **Job controller:** Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
3. **Endpoints controller:** Populates the Endpoints object (that is, joins Services & Pods).
4. **Service Account & Token controllers:** Create default accounts and API access tokens for new namespaces.
5. **etc**

The Controllers
• The controllers are the brain behind orchestration.

• They are responsible for noticing and responding when nodes, containers or endpoints go down.

• The controllers make decisions to bring up new containers in such cases

Kube Cloud-controller-manager (c-c-m)

◇ A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

◇ The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

◇ As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:
1. **Node controller:** For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
2. **Route controller:** For setting up routes in the underlying cloud infrastructure
3. **Service controller:** For creating, updating and deleting cloud provider load balancers
4. **etc**

Node Components

◇ Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Two Node components:
1. Kubelet
2. Kube-proxy
3. Container runtime

kubelet

◇ An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.
◇ The agent is responsible for making sure that the containers are running on the nodes as expected
◇ The kubelet takes a set of PodSpecs that are provided through various mechanisms and

ensures that the containers described in those PodSpecs are running and healthy.
◇ The kubelet doesn't manage containers which were not created by Kubernetes.

kube-proxy

◇ kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

◇ kube-proxy maintains network rules on nodes.

◇ These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

◇ kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

Container runtime

◇ The container runtime is the software that is responsible for running containers.

◇ The container runtime is the underlying software that is used to run containers. In our case it happens to be Docker

• Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

• Is Docker container runtime the default? -- cheek on this also.

Bootstrapping:
• A technique of loading a program into a computer by means of a few initial instructions which enable the introduction of the rest of the program from an input device.

Addons

◇ Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features.

◇ Because these are providing cluster-level features, namespaced resources for addons belong within the kube-system namespace.

DNS

◇ While the other addons are not strictly required, all Kubernetes clusters should have cluster DNS, as many examples rely on it.

◇ Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

◇ Containers started by Kubernetes automatically include this DNS server in their DNS searches.

Web UI (Dashboard)

◇ Dashboard is a general purpose, web-based UI for Kubernetes clusters.

◇ It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

Container Resource Monitoring

◇ Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

Cluster-level Logging

◇ A cluster-level logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

Selected addons are described below; for an extended list of available addons, please see Addons

Reference
https://kubernetes.io/docs/concepts/overview/components/

03_The Kubernetes API

MASTER

The Kubernetes API

• The core of Kubernetes' control plane is the API server.

• The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.

• The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events).

• Most operations can be performed through the kubectl command-line interface or other command-line tools, such as kubeadm, which in turn use the API.

• However, you can also access the API directly using REST calls.

Consider using one of the client libraries if you are writing an application using the Kubernetes API.
https://kubernetes.io/docs/concepts/overview/kubernetes-api/

OpenAPI specification

Complete API details are documented using OpenAPI.

OpenAPI V2

◇ The Kubernetes API server serves an aggregated OpenAPI v2 spec via the `/openapi/v2` endpoint.

◇ Kubernetes implements an alternative Protobuf based serialization format that is primarily intended for intra-cluster communication

OpenAPI V3

◇ Kubernetes v1.24 offers beta support for publishing its APIs as OpenAPI v3; this is a beta feature that is enabled by default beta feature that is enabled by default

◇ A discovery endpoint `/openapi/v3` is provided to see a list of all groups/versions available. This endpoint only returns JSON

Persistence

◇ Kubernetes stores the serialized state of objects by writing them into etcd

API groups and versioning

◇ To make it easier to eliminate fields or restructure resource representations, Kubernetes supports multiple API versions, each at a different API path, such as `/api/v1` or `/apis/rbac.authorization.k8s.io/v1alpha1`.

◇ Versioning is done at the API level rather than at the resource or field level to ensure that the API presents a clear, consistent view of system resources and behavior, and to enable controlling access to end-of-life and/or experimental APIs.

◇ To make it easier to evolve and to extend its API, Kubernetes implements API groups that can be enabled or disabled.

◇ API resources are distinguished by their API group, resource type, namespace (for namespaced resources), and name. The API server handles the conversion between API versions transparently: all the different versions are actually representations of the same persisted data. The API server may serve the same underlying data through multiple API versions

◇ For example, suppose there are two API versions, `v1` and `v1beta1`, for the same resource. If you originally created an object using the `v1beta1` version of its API, you can later read, update, or delete that object using either the `v1beta1` or the `v1` API version.

API changes

◇ Any system that is successful needs to grow and change as new use cases emerge or existing ones change. Therefore, Kubernetes has designed the Kubernetes API to continuously change and grow.

◇ The Kubernetes project aims to *not* break compatibility with existing clients, and to maintain that compatibility for a length of time so that other projects have an opportunity to adapt

In general, new API resources and new resource fields can be added often and frequently. Elimination of resources or fields requires following the API deprecation policy
◇ Kubernetes makes a strong commitment to maintain compatibility for official Kubernetes APIs once they reach general availability (GA), typically at API version `v1`. Additionally, Kubernetes keeps compatibility even for *beta* API versions wherever feasible: if you adopt a beta API you can continue to interact with your cluster using that API, even after the feature goes stable

◇ Although Kubernetes also aims to maintain compatibility for *alpha* APIs versions, in some circumstances this is not possible. If you use any alpha API versions, check the release notes for Kubernetes when upgrading your cluster, in case the API did change.

API Extension

The Kubernetes API can be extended in one of two ways:
1. Custom resources let you declaratively define how the API server should provide your chosen resource API.

2. You can also extend the Kubernetes API by implementing an aggregation layer

Persistence

◇ Kubernetes stores the serialized state of objects by writing them into etcd.

Further reference
https://kubernetes.io/docs/concepts/overview/kubernetes-api/

04_Working with Kubernetes Objects

MASTER

Understanding Kubernetes objects

Kubernetes objects
• *Kubernetes objects* are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

◇ What containerized applications are running (and on which nodes)

◇ The resources available to those applications

◇ The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

◇ A Kubernetes object is a "record of intent"--once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's *desired state*.

◇ To work with Kubernetes objects--whether to create, modify, or delete them--you'll need to use the Kubernetes API. When you use the `kubectl` command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. You can also use the Kubernetes API directly in your own programs using one of the Client Libraries.

Object Spec and Status
◇ Almost every Kubernetes object includes two nested object fields that govern the object's configuration:

1. the object *spec* and

2. the object *status*.

3. For objects that have a spec, you have to set this when you create the object, providing a

description of the characteristics you want the resource to have: its *desired state.*

4. *The status describes the current state of the object, supplied and updated by the Kubernetes system and its components.*

5. *The Kubernetes control plane continually and actively manages every object's actual state to match the desired state you supplied*

For example: in Kubernetes, a Deployment is an object that can represent an application running on your cluster. When you create the Deployment, you might set the Deployment spec to specify that you want three replicas of the application to be running. The Kubernetes system reads the Deployment spec and starts three instances of your desired application--updating the status to match your spec. If any of those instances should fail (a status change), the Kubernetes system responds to the difference between spec and status by making a correction--in this case, starting a replacement instance

Describing a Kubernetes object

◇ *When you create an object in Kubernetes, you must provide the object spec that describes its desired state, as well as some basic information about the object (such as a name).*

◇ *When you use the Kubernetes API to create the object (either directly or via kubectl), that API request must include that information as JSON in the request body.*

◇ ***Most often, you provide the information to kubectl in a .yaml file.*** *kubectl converts the information to JSON when making the API request.*

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

◇ *One way to create a Deployment using a .yaml file like the one above is to use the [kubectl apply]() command in the kubectl command-line interface, passing the .yaml file as an argument. Here's an example:*

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

The output is similar to this:

```
deployment.apps/nginx-deployment created
```

Required Fields

In the .yaml file for the Kubernetes object you want to create, you'll need to set values for the following fields:

1. *apiVersion - Which version of the Kubernetes API you're using to create this object*

2. *kind - What kind of object you want to create*

3. *metadata - Data that helps uniquely identify the object, including a name string, UID, and optional namespace*

4. spec - What state you desire for the object

◇ The precise format of the object spec is different for every Kubernetes object, and contains nested fields specific to that object.

◇ The Kubernetes API Reference can help you find the spec format for all of the objects you can create using Kubernetes.

For example, see the spec field for the Pod API reference. For each Pod, the .spec field specifies the pod and its desired state (such as the container image name for each container within that pod). Another example of an object specification is the spec field for the StatefulSet API. For StatefulSet, the .spec field specifies the StatefulSet and its desired state. Within the .spec of a StatefulSet is a template for Pod objects. That template describes Pods that the StatefulSet controller will create in order to satisfy the StatefulSet specification. Different kinds of object can also have different .status; again, the API reference pages detail the structure of that .status field, and its content for each different type of object.

Kubernetes Object Management

Kubernetes Object Management

The kubectl command-line tool supports several different ways to create and manage Kubernetes objects

Management techniques

1. Imperative commands

2. Imperative object configuration

3. Declarative object configuration

Warning: A Kubernetes object should be managed using only one technique. Mixing and matching techniques for the same object results in undefined behavior.

Imperative commands

• When using imperative commands, a user operates directly on live objects in a cluster. The user provides operations to the kubectl command as arguments or flags.

◇ This is the recommended way to get started or to run a one-off task in a cluster. Because this technique operates directly on live objects, it provides no history of previous configurations.

Examples

Run an instance of the nginx container by creating a Deployment object:

```
kubectl create deployment nginx --image nginx
```

Trade-offs

Advantages compared to object configuration:

◇ Commands are expressed as a single action word.

◇ Commands require only a single step to make changes to the cluster.

Disadvantages compared to object configuration:

◇ Commands do not integrate with change review processes.

◇ Commands do not provide an audit trail associated with changes.

◇ Commands do not provide a source of records except for what is live.

◇ Commands do not provide a template for creating new objects.

Imperative object configuration

◇ In imperative object configuration, the kubectl command specifies the operation (create, replace, etc.), optional flags and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.

◇ **Warning:** The imperative `replace` command replaces the existing spec with the newly provided one, dropping all changes to the object missing from the configuration file. This approach should not be used with resource types whose specs are updated independently of the configuration file. Services of type `LoadBalancer`, for example, have their `externalIPs` field updated independently from the configuration by the cluster.

Examples

Create the objects defined in a configuration file:

```
kubectl create -f nginx.yaml
```

Delete the objects defined in two configuration files:

```
kubectl delete -f nginx.yaml -f redis.yaml
```

Update the objects defined in a configuration file by overwriting the live configuration:

```
kubectl replace -f nginx.yaml
```

Trade-offs

Advantages compared to imperative commands:

◇ Object configuration can be stored in a source control system such as Git.

◇ Object configuration can integrate with processes such as reviewing changes before push and audit trails.

◇ Object configuration provides a template for creating new objects.

Disadvantages compared to imperative commands:

◇ Object configuration requires basic understanding of the object schema.

◇ Object configuration requires the additional step of writing a YAML file.

Advantages compared to declarative object configuration:

◇ Imperative object configuration behavior is simpler and easier to understand.

◇ As of Kubernetes version 1.5, imperative object configuration is more mature.

Disadvantages compared to declarative object configuration:

◇ Imperative object configuration works best on files, not directories.

◇ Updates to live objects must be reflected in configuration files, or they will be lost during the next replacement.

Declarative object configuration

◇ When using declarative object configuration, a user operates on object configuration files stored locally, however the user does not define the operations to be taken on the files.

◇ Create, update, and delete operations are automatically detected per-object by kubectl.

◇ This enables working on directories, where different operations might be needed for different objects.

Note: Declarative object configuration retains changes made by other writers, even if the changes are not merged back to the object configuration file. This is possible by using the `patch` API operation to write only observed differences, instead of using the `replace` API operation to replace the entire object configuration.

Examples

Process all object configuration files in the `configs` directory, and create or patch the live objects. You can first `diff` to see what changes are going to be made, and then apply:

```
kubectl diff -f configs/
```

```
kubectl apply -f configs/
```

Recursively process directories:

```
kubectl diff -R -f configs/
```

```
kubectl apply -R -f configs/
```

Trade-offs

Advantages compared to imperative object configuration:

◇ Changes made directly to live objects are retained, even if they are not merged back into the configuration files.

◇ Declarative object configuration has better support for operating on directories and automatically detecting operation types (create, patch, delete) per-object.

Disadvantages compared to imperative object configuration:

◇ Declarative object configuration is harder to debug and understand results when they are unexpected.

◇ Partial updates using diffs create complex merge and patch operations.

06_Object Names and IDs

Object Names and IDs

• Each object in your cluster has a *Name* that is unique for that type of resource.

• Every Kubernetes object also has a *UID* that is unique across your whole cluster.

◇ For example, you can only have one Pod named `myapp-1234` within the same namespace, but you can have one Pod and one Deployment that are each named `myapp-1234`.

◇ For non-unique user-provided attributes, Kubernetes provides labels and annotations .

Names

◇ A client-provided string that refers to an object in a resource URL, such as `/api/v1/pods/some-name`

◇ Only one object of a given kind can have a given name at a time. However, if you delete the object, you can make a new object with the same name.

Note: In cases when objects represent a physical entity, like a Node representing a physical host, when the host is re-created under the same name without deleting and re-creating the Node, Kubernetes treats the new host as the old one, which may lead to inconsistencies.

Below are four types of commonly used name constraints for resources.

1.
DNS Subdomain Names

2.
RFC 1123 Label Names

3.
RFC 1035 Label Names

4.
Path Segment Names

DNS Subdomain Names

Most resource types require a name that can be used as a DNS subdomain name as defined in RFC 1123. This means the name must:

◇ contain no more than 253 characters

◇ contain only lowercase alphanumeric characters, '-' or '.'

◇ start with an alphanumeric character

◇ end with an alphanumeric character

RFC 1123 Label Names

Some resource types require their names to follow the DNS label standard as defined in RF-C 1123. This means the name must:

◇ contain at most 63 characters

◇ contain only lowercase alphanumeric characters or '-'

◇ start with an alphanumeric character

◇ end with an alphanumeric character

RFC 1035 Label Names

Some resource types require their names to follow the DNS label standard as defined in RF-C 1035. This means the name must:

◇ contain at most 63 characters

◇ contain only lowercase alphanumeric characters or '-'

◇ start with an alphabetic character

◇ end with an alphanumeric character

Path Segment Names

◇ Some resource types require their names to be able to be safely encoded as a path segment.

◇ In other words, the name may not be "." or ".." and the name may not contain "/" or "%".

Here's an example manifest for a Pod named nginx-demo.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Note: Some resource types have additional restrictions on their names.

UIDs

◇ A Kubernetes systems-generated string to uniquely identify objects.

◇ Every object created over the whole lifetime of a Kubernetes cluster has a distinct UID.

◇ It is intended to distinguish between historical occurrences of similar entities.

◇ Kubernetes UIDs are universally unique identifiers (also known as UUIDs). UUIDs are standardized as ISO/IEC 9834-8 and as ITU-T X.667.

Namespaces

Namespaces:

• In Kubernetes, *namespaces* provide a mechanism for isolating groups of resources within a single cluster.

• Names of resources need to be unique within a namespace, but not across namespaces.

• Namespace-based scoping is applicable only for namespaced objects *(e.g. Deployments, Services, etc)* and not for cluster-wide objects *(e.g. StorageClass, Nodes, PersistentVolumes, etc)*.

When to Use Multiple Namespaces

◇ Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

◇ Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.

◇ Namespaces are a way to divide cluster resources between multiple users (via resource quota).

◇ It is not necessary to use multiple namespaces to separate slightly different resources, such as different versions of the same software: use labels to distinguish resources within the same namespace.

Working with Namespaces

Creation and deletion of namespaces are described in the Admin Guide documentation for namespaces.

Note: Avoid creating namespaces with the prefix `kube-`, since it is reserved for Kubernetes system namespaces.

Viewing namespaces

You can list the current namespaces in a cluster using:

```
kubectl get namespace
```

```
NAME              STATUS   AGE
default           Active   1d
kube-node-lease   Active   1d
kube-public       Active   1d
kube-system       Active   1d
```

Kubernetes starts with four initial namespaces:

1. Default

2. Kube-system

3. Kube-public

4. Kube-node-lease

◇ default The default namespace for objects with no other namespace

◇ kube-system The namespace for objects created by the Kubernetes system

◇ kube-public This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

◇ kube-node-lease This namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane can detect node failure.

Setting the namespace for a request

To set the namespace for a current request, use the `--namespace` flag

For example:

```
kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
kubectl get pods --namespace=<insert-namespace-name-here>
```

Setting the namespace preference

You can permanently save the namespace for all subsequent kubectl commands in that context.

```
kubectl config set-context --current --namespace=<insert-namespace-name-here>

# Validate it

kubectl config view --minify | grep namespace:
```

Namespaces and DNS

When you create a Service, it creates a corresponding DNS entry.
This entry is of the form `<service-name>.<namespace-name>.svc.cluster.local`, which means that if a container only uses `<service-name>`, it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).
As a result, all namespace names must be valid RFC 1123 DNS labels.

Warning:
By creating namespaces with the same name as public top-level domains, Services in these namespaces can have short DNS names that overlap with public DNS records. Workloads from any namespace performing a DNS lookup without a trailing dot will be redirected to those services, taking precedence over public DNS.
To mitigate this, limit privileges for creating namespaces to trusted users. If required, you

could additionally configure third-party security controls, such as admission webhooks, to block creating any namespace with the name of public TLDs

Not All Objects are in a Namespace

◇ Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. However namespace resources are not themselves in a namespace. And low-level resources, such as nodes and persistentVolumes, are not in any namespace.

To see which Kubernetes resources are and aren't in a namespace:

```
# In a namespace
kubectl api-resources --namespaced=true

# Not in a namespace
kubectl api-resources --namespaced=false
```

Automatic labelling

FEATURE STATE: Kubernetes 1.21 [beta]

◇ The Kubernetes control plane sets an immutable label kubernetes.io/metadata.name on all namespaces, provided that the NamespaceDefaultLabelName feature gate is enabled. The value of the label is the namespace name

Kubernetes starts with four initial namespaces:

◇ default The default namespace for objects with no other namespace

◇ kube-system The namespace for objects created by the Kubernetes system

◇ kube-public This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

◇ kube-node-lease This namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane can detect node failure.

◇

Setting the namespace for a request

To set the namespace for a current request, use the --namespace flag.
For example:

```
kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
```

```
kubectl get pods --namespace=<insert-namespace-name-here>
```

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

08_Labels and Selectors

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

02_Cluster_Architecture

01_Nodes

https://kubernetes.io/docs/concepts/architecture/nodes/

Nodes

• A node may be a virtual or physical machine, depending on the cluster.
• Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have only one node.
• Each node is managed by the control plane and contains the services necessary to run P-ods.
• Kubernetes runs your workload by placing containers into Pods to run on *Nodes*.
• The components on a node include the

1. kubelet,

2. container runtime

3. kube-proxy.

Management

There are two main ways to have Nodes added to the API server:
1. The kubelet on a node self-registers to the control plane

2. You (or another human user) manually add a Node object

Self-registration of a node to cluster

After you create a Node object, or the kubelet on a node self-registers, the control plane checks whether the new Node object is valid. For example, if you try to create a Node from the following JSON manifest:

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

◇ Kubernetes creates a Node object internally (the representation). Kubernetes checks that a kubelet has registered to the API server that matches the metadata.name field of the Node. If the node is healthy (i.e. all necessary services are running), then it is eligible to run a Pod. Otherwise, that node is ignored for any cluster activity until it becomes healthy.

Note:
◇ Kubernetes keeps the object for the invalid Node and continues checking to see whether it becomes healthy.

◇ You, or a controller, must explicitly delete the Node object to stop that health checking.

The name of a Node object must be a valid DNS subdomain name.

Node name uniqueness

◇ The name identifies a Node.

◇ Two Nodes cannot have the same name at the same time.

◇ Kubernetes also assumes that a resource with the same name is the same object.

◇ In case of a Node, it is implicitly assumed that an instance using the same name will have the same state (e.g. network settings, root disk contents) and attributes like node labels.

◇ This may lead to inconsistencies if an instance was modified without changing its name. If the Node needs to be replaced or updated significantly, the existing Node object needs to be removed from the API server first and re-added after the update.

Self-registration of Nodes

When the kubelet flag --register-node is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.
For self-registration, the kubelet is started with the following options:
◇ --kubeconfig - Path to credentials to authenticate itself to the API server.

◇ --cloud-provider - How to talk to a cloud provider to read metadata about itself.

◇ --register-node - Automatically register with the API server.

◇ --register-with-taints - Register the node with the given list of taints (comma separated <key>=<value>:<effect>).
No-op if register-node is false.

◇ --node-ip - IP address of the node.

◇ --node-labels - Labels to add when registering the node in the cluster (see label restrictions enforced by the NodeRestriction admission plugin).

◇ --node-status-update-frequency - Specifies how often kubelet posts its node status to the API server.

When the Node authorization mode and NodeRestriction admission plugin are enabled, kubelets are only authorized to create/modify their own Node resource.

Note:
As mentioned in the Node name uniqueness section, when Node configuration needs to be

updated, it is a good practice to re-register the node with the API server. For example, if the kubelet is restarted with the new set of --node-labels, but the same Node name is used, the change will not take effect, as labels are being set on the Node registration.
Pods already scheduled on the Node may misbehave or cause issues if the Node configuration will be changed on kubelet restart. For example, already running Pod may be tainted against the new labels assigned to the Node, while other Pods, that are incompatible with that Pod will be scheduled based on this new label. Node re-registration ensures all Pods will be drained and properly re-scheduled.

Manual Node administration

◇ You can create and modify Node objects using kubectl.

◇ When you want to create Node objects manually, set the kubelet flag --register-node=false.

◇ You can modify Node objects regardless of the setting of --register-node. For example, you can set labels on an existing Node or mark it unschedulable.

◇ You can use labels on Nodes in conjunction with node selectors on Pods to control scheduling. For example, you can constrain a Pod to only be eligible to run on a subset of the available nodes.

Marking a node as unschedulable prevents the scheduler from placing new pods onto that Node but does not affect existing Pods on the Node. This is useful as a preparatory step before a node reboot or other maintenance.
To mark a Node unschedulable, run:

```
kubectl cordon $NODENAME
```

Note: Pods that are part of a DaemonSet tolerate being run on an unschedulable Node. DaemonSets typically provide node-local services that should run on the Node even if it is being drained of workload applications.

Node status

A Node's status contains the following information:

1. Addresses

2. Conditions

3. Capacity and Allocatable

4. Info

◇ You can use kubectl to view a Node's status and other details:

```
< kubectl describe node <insert-node-name-
here> >
```

Each section of the output is described below.

Addresses

The usage of these fields varies depending on your cloud provider or bare metal configuration.

◇ HostName: The hostname as reported by the node's kernel. Can be overridden via the kubelet --hostname-override parameter.

◇ ExternalIP: Typically the IP address of the node that is externally routable (available from outside the cluster).

◇ InternalIP: Typically the IP address of the node that is routable only within the cluster.

◇

Conditions

The conditions field describes the status of all Running nodes. Examples of conditions include:

Node Condition	Description
Ready	True if the node is healthy a node is not healthy and is n the node controller has not monitor-grace-period (defau
DiskPressure	True if pressure exists on th capacity is low; otherwise F
MemoryPressure	True if pressure exists on th memory is low; otherwise F
PIDPressure	True if pressure exists on th many processes on the nod
NetworkUnavailable	True if the network for the r otherwise False

Note: If you use command-line tools to print details of a cordoned Node, the Condition includes SchedulingDisabled. SchedulingDisabled is not a Condition in the Kubernetes API; instead, cordoned nodes are marked Unschedulable in their spec.

In the Kubernetes API, a node's condition is represented as part of the .status of the Node resource. For example, the following JSON structure describes a healthy node:

```
"conditions": [
  {
    "type": "Ready",
    "status": "True",
    "reason": "KubeletReady",
    "message": "kubelet is posting ready status",
    "lastHeartbeatTime": "2019-06-05T18:38:35Z",
    "lastTransitionTime": "2019-06-05T11:41:27Z"
  }
]
```

If the status of the Ready condition remains Unknown or False for longer than the pod-eviction-timeout (an argument passed to the kube-controller-manager), then the node controller triggers API-initiated eviction for all Pods assigned to that node. The default eviction timeout duration is **five minutes**. In some cases when the node is unreachable, the API server is unable to communicate with the kubelet on the node. The decision to delete the pods cannot be communicated to the kubelet until communication with the API server is re-established. In the meantime, the pods that are scheduled for deletion may continue to run on the partitioned node.
The node controller does not force delete pods until it is confirmed that they have stopped running in the cluster. You can see the pods that might be running on an unreachable node as being in the Terminating or Unknown state. In cases where Kubernetes cannot deduce from the underlying infrastructure if a node has permanently left a cluster, the cluster administrator may need to delete the node object by hand. Deleting the node object from Kubernetes causes all the Pod objects running on the node to be deleted from the API server and frees up their names.
When problems occur on nodes, the Kubernetes control plane automatically creates taints that match the conditions affecting the node. The scheduler takes the Node's taints into consideration when assigning a Pod to a Node. Pods can also have tolerations that let them run on a Node even though it has a specific taint.
See Taint Nodes by Condition for more details.

Capacity and Allocatable

Describes the resources available on the node: CPU, memory, and the maximum number of pods that can be scheduled onto the node.

The fields in the capacity block indicate the total amount of resources that a Node has. The allocatable block indicates the amount of resources on a Node that is available to be consumed by normal Pods.

You may read more about capacity and allocatable resources while learning how to [reserve compute resources]() on a Node.

Info

Describes general information about the node, such as kernel version, Kubernetes version (kubelet and kube-proxy version), container runtime details, and which operating system the node uses. The kubelet gathers this information from the node and publishes it into the Kubernetes API.

Heartbeats

Heartbeats, sent by Kubernetes nodes, help your cluster determine the availability of each node, and to take action when failures are detected.

For nodes there are two forms of heartbeats:

◇ updates to the `.status` of a Node

◇ [Lease]() objects within the `kube-node-lease` [namespace](). Each Node has an associated Lease object.

Compared to updates to `.status` of a Node, a Lease is a lightweight resource. Using Leases for heartbeats reduces the performance impact of these updates for large clusters.

The kubelet is responsible for creating and updating the `.status` of Nodes, and for updating their related Leases.

◇ The kubelet updates the node's `.status` either when there is change in status or if there has been no update for a configured interval. The default interval for `.status` updates to Nodes is 5 minutes, which is much longer than the 40 second default timeout for unreachable nodes.

◇ The kubelet creates and then updates its Lease object every 10 seconds (the default update interval). Lease updates occur independently from updates to the Node's `.status`. If the Lease update fails, the kubelet retries, using exponential backoff that starts at 200 milliseconds and capped at 7 seconds.

Node controller

The node controller is a Kubernetes control plane component that manages various aspects of nodes.
The node controller has multiple roles in a node's life. The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).
The second is keeping the node controller's internal list of nodes up to date with the cloud provider's list of available machines. When running in a cloud environment and whenever a node is unhealthy, the node controller asks the cloud provider if the VM for that node is still available. If not, the node controller deletes the node from its list of nodes.
The third is monitoring the nodes' health. The node controller is responsible for:
◇ In the case that a node becomes unreachable, updating the Ready condition in the Node's .status field. In this case the node controller sets the Ready condition to Unknown.

◇ If a node remains unreachable: triggering API-initiated eviction for all of the Pods on the unreachable node. By default, the node controller waits 5 minutes between marking the node as Unknown and submitting the first eviction request.

By default, the node controller checks the state of each node every 5 seconds. This period can be configured using the --node-monitor-period flag on the kube-controller-manager component.

Rate limits on eviction

In most cases, the node controller limits the eviction rate to --node-eviction-rate (default 0.1) per second, meaning it won't evict pods from more than 1 node per 10 seconds.
The node eviction behavior changes when a node in a given availability zone becomes unhealthy. The node controller checks what percentage of nodes in the zone are unhealthy (the Ready condition is Unknown or False) at the same time:
◇ If the fraction of unhealthy nodes is at least --unhealthy-zone-threshold (default 0.55), then the eviction rate is reduced.

◇ If the cluster is small (i.e. has less than or equal to --large-cluster-size-threshold nodes - default 50), then evictions are stopped.

◇ Otherwise, the eviction rate is reduced to --secondary-node-eviction-rate (default 0.01) per second.

The reason these policies are implemented per availability zone is because one availability zone might become partitioned from the control plane while the others remain connected. If your cluster does not span multiple cloud provider availability zones, then the eviction mechanism does not take per-zone unavailability into account.
A key reason for spreading your nodes across availability zones is so that the workload can be shifted to healthy zones when one entire zone goes down. Therefore, if all nodes in a zone are unhealthy, then the node controller evicts at the normal rate of --node-eviction-rate. The corner case is when all zones are completely unhealthy (none of the nodes in the cluster are healthy). In such a case, the node controller assumes that there is some problem with connectivity between the control plane and the nodes, and doesn't perform any evictions. (If there has been an outage and some nodes reappear, the node controller does evict pods from the remaining nodes that are unhealthy or unreachable).

The node controller is also responsible for evicting pods running on nodes with NoExecute taints, unless those pods tolerate that taint. The node controller also adds taints corresponding to node problems like node unreachable or not ready. This means that the scheduler won't place Pods onto unhealthy nodes.

Resource capacity tracking

Node objects track information about the Node's resource capacity: for example, the amount of memory available and the number of CPUs. Nodes that self register report their capacity during registration. If you manually add a Node, then you need to set the node's capacity information when you add it.
The Kubernetes scheduler ensures that there are enough resources for all the Pods on a Node. The scheduler checks that the sum of the requests of containers on the node is no greater than the node's capacity. That sum of requests includes all containers managed by the kubelet, but excludes any containers started directly by the container runtime, and also excludes any processes running outside of the kubelet's control.

Note: If you want to explicitly reserve resources for non-Pod processes, see reserve resources for system daemons.

Node topology

FEATURE STATE: Kubernetes v1.18 [beta]

If you have enabled the TopologyManager feature gate, then the kubelet can use topology hints when making resource assignment decisions. See Control Topology Management Policies on a Node for more information.

Graceful node shutdown

FEATURE STATE: Kubernetes v1.21 [beta]

The kubelet attempts to detect node system shutdown and terminates pods running on the node.

Kubelet ensures that pods follow the normal pod termination process during the node shutdown.

The Graceful node shutdown feature depends on systemd since it takes advantage of systemd inhibitor locks to delay the node shutdown with a given duration.

Graceful node shutdown is controlled with the GracefulNodeShutdown feature gate which is enabled by default in 1.21.

Note that by default, both configuration options described below, shutdownGracePeriod and shutdownGracePeriodCriticalPods are set to zero, thus not activating the graceful node shutdown functionality. To activate the feature, the two kubelet config settings should be configured appropriately and set to non-zero values.

During a graceful shutdown, kubelet terminates pods in two phases:

1. Terminate regular pods running on the node.

2. Terminate critical pods running on the node.

Graceful node shutdown feature is configured with two KubeletConfiguration options:

◇ shutdownGracePeriod:

■ Specifies the total duration that the node should delay the shutdown by. This is the total grace period for pod termination for both regular and critical pods.

◇ shutdownGracePeriodCriticalPods:

■ Specifies the duration used to terminate critical pods during a node shutdown. This value should be less than shutdownGracePeriod.

For example, if `shutdownGracePeriod=30s`, and `shutdownGracePeriodCriticalPods=10s`, kubelet will delay the node shutdown by 30 seconds. During the shutdown, the first 20 (30-10) seconds would be reserved for gracefully terminating normal pods, and the last 10 seconds would be reserved for terminating critical pods.

Note:

When pods were evicted during the graceful node shutdown, they are marked as shutdown. Running `kubectl get pods` shows the status of the the evicted pods as `Terminated`. And `kubectl describe pod` indicates that the pod was evicted because of node shutdown:

```
Reason:       Terminated
Message:      Pod was terminated in response to imminent node shutdown.
```

Non Graceful node shutdown

FEATURE STATE: Kubernetes v1.24 [alpha]

A node shutdown action may not be detected by kubelet's Node Shutdown Manager, either because the command does not trigger the inhibitor locks mechanism used by kubelet or because of a user error, i.e., the ShutdownGracePeriod and ShutdownGracePeriodCriticalPods are not configured properly. Please refer to above section Graceful Node Shutdown for more details.

When a node is shutdown but not detected by kubelet's Node Shutdown Manager, the pods that are part of a StatefulSet will be stuck in terminating status on the shutdown node and cannot move to a new running node. This is because kubelet on the shutdown node is not available to delete the pods so the StatefulSet cannot create a new pod with the same name. If there are volumes used by the pods, the VolumeAttachments will not be deleted from the original shutdown node so the volumes used by these pods cannot be attached to a new running node. As a result, the application running on the StatefulSet cannot function properly. If the original shutdown node comes up, the pods will be deleted by kubelet and new pods will be created on a different running node. If the original shutdown node does not come up,
these pods will be stuck in terminating status on the shutdown node forever.

To mitigate the above situation, a user can manually add the taint `node kubernetes.io/out-of-service` with either `NoExecute` or `NoSchedule` effect to a Node marking it out-of-service. If the `NodeOutOfServiceVolumeDetach`feature gate is enabled on `kube-controller-manager`, and a Node is marked out-of-service with this taint, the pods on the node will be forcefully deleted if there are no matching tolerations on it and volume detach operations for the pods terminating on the node will happen immediately. This allows the Pods on the out-of-service node to recover quickly on a different node.

During a non-graceful shutdown, Pods are terminated in the two phases:
1. Force delete the Pods that do not have matching out-of-service tolerations.

2. Immediately perform detach volume operation for such pods.

Note:
◇ Before adding the taint node.kubernetes.io/out-of-service , it should be verified that the node is already in shutdown or power off state (not in the middle of restarting).

◇ The user is required to manually remove the out-of-service taint after the pods are moved to a new node and the user has checked that the shutdown node has been recovered since the user was the one who originally added the taint.

◇

Pod Priority based graceful node shutdown

FEATURE STATE: Kubernetes v1.23 [alpha]
To provide more flexibility during graceful node shutdown around the ordering of pods during shutdown, graceful node shutdown honors the PriorityClass for Pods, provided that you enabled this feature in your cluster. The feature allows cluster administers to explicitly define the ordering of pods during graceful node shutdown based on priority classes.
The Graceful Node Shutdown feature, as described above, shuts down pods in two phases, non-critical pods, followed by critical pods. If additional flexibility is needed to explicitly define the ordering of pods during shutdown in a more granular way, pod priority based graceful shutdown can be used.
When graceful node shutdown honors pod priorities, this makes it possible to do graceful node shutdown in multiple phases, each phase shutting down a particular priority class of pods. The kubelet can be configured with the exact phases and shutdown time per phase.

Assuming the following custom pod priority classes in a cluster,

Pod priority class name	Pod priority class value
custom-class-a	100000
custom-class-b	10000
custom-class-c	1000
regular/unset	0

Within the kubelet configuration the settings for shutdownGracePeriodByPodPriority could look like:

Pod priority class value	Shutdown period
100000	10 seconds
10000	180 seconds
1000	120 seconds
0	60 seconds

The corresponding kubelet config YAML configuration would be:

```
shutdownGracePeriodByPodPriority:

- priority: 100000

  shutdownGracePeriodSeconds: 10

- priority: 10000

  shutdownGracePeriodSeconds: 180

- priority: 1000

  shutdownGracePeriodSeconds: 120

- priority: 0

  shutdownGracePeriodSeconds: 60
```

The above table implies that any pod with priority value >= 100000 will get just 10 seconds to stop, any pod with value >= 10000 and < 100000 will get 180 seconds to stop, any pod with value >= 1000 and < 10000 will get 120 seconds to stop. Finally, all other pods will get 60 seconds to stop.
One doesn't have to specify values corresponding to all of the classes. For example, you could instead use these settings:

Pod priority class value	**Shutdown period**
100000	300 seconds
1000	120 seconds
0	60 seconds

In the above case, the pods with custom-class-b will go into the same bucket as custom-class-c for shutdown.
If there are no pods in a particular range, then the kubelet does not wait for pods in that priority range. Instead, the kubelet immediately skips to the next priority class value range.
If this feature is enabled and no configuration is provided, then no ordering action will be taken.
Using this feature requires enabling the GracefulNodeShutdownBasedOnPodPriority feature gate , and setting ShutdownGracePeriodByPodPriority in the kubelet config to the desired configuration containing the pod priority class values and their respective shutdown periods.
Note: The ability to take Pod priority into account during graceful node shutdown was introduced as an Alpha feature in Kubernetes v1.23. In Kubernetes 1.24 the feature is Beta and is enabled by default.
Metrics graceful_shutdown_start_time_seconds and graceful_shutdown_end_time_seconds are emitted under the kubelet subsystem to monitor node shutdowns.

Swap memory management

FEATURE STATE: Kubernetes v1.22 [alpha]
Prior to Kubernetes 1.22, nodes did not support the use of swap memory, and a kubelet would by default fail to start if swap was detected on a node. In 1.22 onwards, swap memory support can be enabled on a per-node basis.
To enable swap on a node, the NodeSwap feature gate must be enabled on the kubelet, and the --fail-swap-on command line flag or failSwapOn configuration setting must be set to false.
Warning: When the memory swap feature is turned on, Kubernetes data such as the

content of Secret objects that were written to tmpfs now could be swapped to disk.
A user can also optionally configure memorySwap.swapBehavior in order to specify how a node will use swap memory. For example,

```
memorySwap:

 swapBehavior: LimitedSwap
```

The available configuration options for swapBehavior are:

◇ LimitedSwap: Kubernetes workloads are limited in how much swap they can use. Workloads on the node not managed by Kubernetes can still swap.

◇ UnlimitedSwap: Kubernetes workloads can use as much swap memory as they request, up to the system limit.

If configuration for memorySwap is not specified and the feature gate is enabled, by default the kubelet will apply the same behaviour as the LimitedSwap setting.
The behaviour of the LimitedSwap setting depends if the node is running with v1 or v2 of control groups (also known as "cgroups"):

◇ **cgroupsv1:** Kubernetes workloads can use any combination of memory and swap, up to the pod's memory limit, if set.

◇ **cgroupsv2:** Kubernetes workloads cannot use swap memory.

For more information, and to assist with testing and provide feedback, please see KEP-2400 and its design proposal.

02_Communication between Nodes and the Control Plane

Communication between Nodes and the Control Plane

This document catalogs the communication paths between the API server and the Kubernetes cluster. The intent is to allow users to customize their installation to harden the network configuration such that the cluster can be run on an untrusted network (or on fully public IPs on a cloud provider).

Node to Control Plane

Kubernetes has a "hub-and-spoke" API pattern. All API usage from nodes (or the pods they run) terminates at the API server. None of the other control plane components are designed to expose remote services. The API server is configured to listen for remote connections on a secure HTTPS port (typically 443) with one or more forms of client authe-ntication enabled. One or more forms of authorization should be enabled, especially if ano-nymous requests or service account tokens are allowed.

Nodes should be provisioned with the public root certificate for the cluster such that they can connect securely to the API server along with valid client credentials. A good approach is that the client credentials provided to the kubelet are in the form of a client certificate. See kubelet TLS bootstrapping for automated provisioning of kubelet client certificates.

Pods that wish to connect to the API server can do so securely by leveraging a service account so that Kubernetes will automatically inject the public root certificate and a valid bearer token into the pod when it is instantiated. The `kubernetes` service (in `default`

namespace) is configured with a virtual IP address that is redirected (via kube-proxy) to the HTTPS endpoint on the API server.
The control plane components also communicate with the API server over the secure port.
As a result, the default operating mode for connections from the nodes and pods running on the nodes to the control plane is secured by default and can run over untrusted and/or public networks.

Control plane to node

There are two primary communication paths from the control plane (the API server) to the nodes. The first is from the API server to the kubelet process which runs on each node in the cluster. The second is from the API server to any node, pod, or service through the API server's *proxy* functionality.

API server to kubelet

The connections from the API server to the kubelet are used for:

• Fetching logs for pods.

• Attaching (usually through kubectl) to running pods.

• Providing the kubelet's port-forwarding functionality.

These connections terminate at the kubelet's HTTPS endpoint. By default, the API server does not verify the kubelet's serving certificate, which makes the connection subject to man-in-the-middle attacks and **unsafe** to run over untrusted and/or public networks.
To verify this connection, use the --kubelet-certificate-authority flag to provide the API server with a root certificate bundle to use to verify the kubelet's serving certificate.
If that is not possible, use SSH tunneling between the API server and kubelet if required to avoid connecting over an untrusted or public network.
Finally, Kubelet authentication and/or authorization should be enabled to secure the kubelet API.

API server to nodes, pods, and services

The connections from the API server to a node, pod, or service default to plain HTTP connections and are therefore neither authenticated nor encrypted. They can be run over a secure HTTPS connection by prefixing https: to the node, pod, or service name in the API URL, but they will not validate the certificate provided by the HTTPS endpoint nor provide client credentials. So while the connection will be encrypted, it will not provide any guarantees of integrity. These connections **are not currently safe** to run over untrusted or public networks.

SSH tunnels

Kubernetes supports SSH tunnels to protect the control plane to nodes communication paths. In this configuration, the API server initiates an SSH tunnel to each node in the cluster (connecting to the SSH server listening on port 22) and passes all traffic destined for a kubelet, node, pod, or service through the tunnel. This tunnel ensures that the traffic is not exposed outside of the network in which the nodes are running.

Note: SSH tunnels are currently deprecated, so you shouldn't opt to use them unless you know what you are doing. The Konnectivity service is a replacement for this communication channel.

Konnectivity service

FEATURE STATE: Kubernetes v1.18 [beta]

As a replacement to the SSH tunnels, the Konnectivity service provides TCP level proxy for the control plane to cluster communication. The Konnectivity service consists of two parts: the Konnectivity server in the control plane network and the Konnectivity agents in the nodes network. The Konnectivity agents initiate connections to the Konnectivity server and maintain the network connections. After enabling the Konnectivity service, all control plane to nodes traffic goes through these connections.

TEMP_TEMP

01

Master vs Worker Nodes
how does one server become a master and the other slave/worker?
• The master server has the kube-apiserver and that is what makes it a master

0 The master also has the controller manager and the scheduler
0
◇ Similarly, the worker nodes have the kubelet agent that is responsible for interacting with the master to provide health information of the worker node and carry out actions requested by the master on the worker nodes

◇ The worker node (or minion) as it is also known, is where the containers are hosted.

◇ For example, Docker containers, and to run docker containers on a system, we need a container runtime installed.

◇ And that is where the container runtime falls. In this case it happens to be Docker.

◇ This does not HAVE to be docker, there are other container runtime alternatives available such as Rocket or CR10.

◇ But throughout this course we are going to use Docker as our container runtime

◇ All the information gathered are stored in a key-value store on the Master.

◇ The key value store is based on the popular etcd framework as we just discussed kube command line tool or kubectl or kube control

◇ The kube control tool is used to deploy and manage applications on a kubernetes cluster, to get cluster information, get the status of nodes in the cluster and many other things.
Kubectl

Kubecti run hello-minikube

Kubectl cluster-info Kubectl get nodes

•	**kubectl run**
•	kubectl cluster-info
•	kubectl get pod

Setup Kubernetes
◇ There are lots of ways to setup Kuberentes. We can setup it up ourselves locally on our laptops or virtual machines using solutions like Minikube and Kubeadmin.

◇ Minikube is a tool used to setup a single instance of Kubernetes in an All-in-one setup and

◇ kubeadmin is a tool used to configure kubernetes in a multi-node setup.

◇ There are also hosted solutions available for setting up kubernetes in a cloud environment such as GCP and AWS

◇ check out play-with-k8s.com

Minikube
◇ Minikube which is the easiest way to get started with Kubernetes on a local system

◇ Earlier we talked about the different components of Kubernetes that make up a Master and worker nodes such as the alai server, etcd key value store, controllers, and scheduler on the master and kubelets and container runtime on the worker nodes.

◇ It would take a lot of time and effort to setup and install all these various components on different systems individually by ourselves.

◇ Minikube bundles all of these different components into a single image providing us a pre-
configured single node kubernetes cluster so we can get started in a matter of minutes.

◇ The whole bundle is packaged into an ISO image and is available online for download

◇ And finally, to interact with the kubernetes cluster, you must have the kubectl kubernetes command line tool also installed on your machine

◇ So you need 3 things to get this working,

◇ you must have a hypervisor installed,

◇ kubectl installed and

◇ minikube executable installed on your system.

Bit doubt on hypervisor

<u>Setup kubeadm</u>

◇ In this lecture we will look at the kubeadm tool which can be used to bootstrap a kubernetes cluster

◇ With the minikube utility you could only setup a single node kubernetes cluster.

◇ The kubeadmin tool helps us setup a multi node cluster with master and workers on separate machines

◇ Installing all of these various components individually on different nodes and modifying the configuration files to make it work is a tedious task. Kubeadmin tool helps us in doing all of that very easily

Setting up steps:

◇ First, you must have multiple systems or virtual machines created for configuring a cluster

◇ Once the systems are created, designate one as master and others as worker nodes.

◇ The next step is to install a container runtime on the hosts.

◇ We will be using Docker, so we must install Docker on all the nodes. (Including the master)

◇ The next step is to install kubeadmin tool on all the nodes.

◇ The kubeadmin tool helps us bootstrap the kubernetes solution by installing and configuring all the required components in the right nodes

◇ The third step is to initialize the Master server.

◇ During this process all the required components are installed and configured on the master server. That way we can start the cluster level configurations from the master server

◇ Once the master is initialized and before joining the worker nodes to the master, we must ensure that the network pre-requisites are met. A normal network connectivity between the systems is not SUFFICIENT for this. Kubernetes requires a special network between the master and worker nodes which is called as a POD network

◇ The last step is to join the worker nodes to the master node. We are then all set to launch our application in the kubernetes environment

POD
◇ At this point, we assume that the application is already developed and built into Docker images, and it is available on a Docker repository like Docker hub, so Kubernetes can pull it down.

◇ We also assume that the Kubernetes cluster has already been set up and is working.

◇ This could be a single-node setup or a multi-node setup, does not matter. All the services need to be in a running state

◇ As we discussed before, with kubernetes our aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster.

◇ kubernetes does not deploy containers directly on the worker nodes.

◇ The containers are encapsulated into a Kubernetes object known as PODs. a A POD is a single instance of an application.

◇ A POD is the smallest object, that you can create in Kubernetes

◇ "A POD is a Kubernetes object that is used to encapsulate a container."

◇ Here we see the simplest of simplest cases were you have a single node kubernetes cluster with a single instance of your application running in a single docker container encapsulated in a POD. What if the number of users accessing your application increase and you need to scale your application? You need to add additional instances of your web application to share the load. Now, were would you spin up additional instances? Do we bring up a new container instance within the same POD? No! We create a new POD altogether with a new instance of the same application. As you can see, we now have two instances of our web application running on two separate PODs on the same kubernetes system or node.

◇ What if the user base FURTHER increases and your current node has no sufficient capacity? Well THEN you can always deploy additional PODs on a new node in the cluster. You will have a new node added to the cluster to expand the cluster's physical capacity. SO, what I am trying to illustrate in this slide is that

◇ PODs usually have a one-to-one relationship with containers running your application.

◇ To scale UP you create new PODs and to scale down you delete PODs.

◇ You do not add additional containers to an existing POD to scale your application.

Multi — Container PODS
◇ Now we just said that PODs usually have a one-to-one relationship with the containers, but, are we restricted to having a single container in a single POD?

a No! A single POD CAN have multiple containers, except for the fact that they are usually not multiple containers of the same kind.

o As we discussed in the previous slide, if our intention was to scale our application, then we would need to create additional PODS
◇ But sometimes you might have a scenario where you have a helper container, that might be doing supporting task for our web application such as processing a user entered data, processing a file uploaded by the user etc. and you want these helper containers to live alongside your application container. In that case, you CAN have both of these container's part of the same POD, so that when a new application container is created, the helper is also created and when it dies the helper also dies since they are part of the same POD. The two containers can also communicate with each other directly by referring to each other as flocalhost' since they share the same network namespace. Plus, they can easily share the same storage space as well

◇ If you still have doubts in this topic (I would understand if you did because I did the first time, I learned these concepts), we could take another shot at understanding PODs from a different angle. Let us, for a moment, keep kubernetes out of our discussion and talk about simple docker containers. Let us assume we were developing a process or a script to deploy our application on a docker host. Then we would first simply deploy our application using a simple docker run python-app command and the application runs fine and our users are able to access it. When the load increases, we deploy more instances of our application by running the docker run commands many more times. This works fine and we are all happy. Now, sometime in the future our application is further developed, undergoes architectural changes, and grows and gets complex. We now have new helper containers that helps our web applications by processing or fetching data from elsewhere. These helper containers maintain a oneto-one relationship with our application container and thus, needs to communicate with the application containers directly and access data from those containers. For this we need to maintain a map of what app and helper containers are connected to each other, we would need to establish network connectivity between these containers ourselves using links and custom networks, we would need to create shareable volumes and share it among the containers and maintain a map of that as well. And most importantly we would need to monitor the state of the application container and when it dies, manually kill the helper container as well as it's no longer required. When a new container is deployed, we would need to deploy the new helper container as well

◇ With PODs, kubernetes does all of this for us automatically.

◇ We just need to define what containers a POD consists of and the containers in a POD by default will have access to the same storage, the same network namespace, and same fate as in they will be created together and destroyed together

◇ Even if our application did not happen to be so complex and we could live with a single container, kubernetes still requires you to create PODs.

◇ But this is good in the long run as your application is now equipped for architectural changes and scale in the future.

◇ However, multi-container pods are a rare use-case, and we are going to stick to single container per POD in this course

KUBECTL
◇ Earlier we learned about the kubectl run command.

◇ What this command really does is it deploys a docker container by creating a POD.

◇ So it first creates a POD automatically and deploys an instance of the nginx docker image.

◇ But where does it get the application image from? For that you need to specify the image name using the —image parameter

Kubectl
Kubectl run nginix —image nginx
Kubectl get pods
◇ The application image, in this case the nginx image, is downloaded from the docker hub repository. Docker hub as we discussed is a public repository were latest docker images of various applications are stored.

◇ You could configure kubernetes to pull the image from the public docker hub or a private repository within the organization

◇ How do we see the list of PODs available?

◇ kubectl get PODs --- command helps us see the list of pods in our cluster

POD with yam!
◇ Kubernetes uses YAML files as input for the creation of objects such as

1. PODs

2. Replicas

3. Deployments

4. Services

5. And etc

A kubernetes definition file always contains 4 top level fields.
1. The apiVersion

2. kind

3. metadata and

4. spec

◇ These are top level or root level properties. Think of them as siblings, children of the same parent. These are all REQUIRED fields, so you MUST have them in your configuration file.

apiVersion
◇ This is the version of the kubernetes API we are using to create the object.

◇ Depending on what we are trying to create we must use the RIGHT apiVersion

◇ For now since we are working on PODs, we will set the apiVersion as vi.

Kind
◇ The kind refers to the type of object we are trying to create, which in this case happens to be a POD. So we will set it as Pod

◇ Some other possible values here could be ReplicaSet or Deployment or Service, which is what you see in the kind field in the table on the right.

<u>Metadata</u>
◇ The metadata is data about the object like its name, labels etc

Need to go through the book on this area.
Once the file is created, run the command
kubectl create -f pod-definition.ymland kubernetes creates the pod
So to summarize remember the 4 top level properties.
apiVersion,
kind, metadata and
spec.
Then start by adding values to those depending on the object you are creating
Commands
Once we create the pod, how do you see it? Use the kubectl get pods command to see a list of pods available. In this case its just one. To see detailed information about the pod run the kubectl describe pod command
This will tell you information about the POD, when it was created, what labels are assigned to it, what docker containers are part of it and the events associated with that POD

Replication Controllers
◇ Kubernetes Controllers are the brain behind Kubernetes.

◇ Controllers monitor kubernetes objects and respond accordingly.

◇ In this lecture we will discuss about one controller. And that is the Replication Controller.

<u>Why do we need a replication controller?</u>
1. High Availability

2. Load Balancing and Scaling

High Availability
Let us go back to our first scenario where we had a single POD running our application. What if for some reason, our application crashes and the POD fails? Users will no longer be able to access our application. To prevent users from losing access to our application, we would like to have more than one instance or POD running at the same time. That way if one fails, we still have our application running on the other one. The replication controller helps us run multiple instances of a single POD in the kubernetes cluster thus providing High Availability.
So, does that mean you cannot use a replication controller if you plan to have a single POD? No! Even if you have a single POD, the replication controller can help by automatically bringing up a new POD when the existing one fails. Thus, the replication controller ensures that the specified number of PODs are always running. Even if it is just 1 or 100

Load Balancing and scaling
Another reason we need replication controller is to create multiple PODs to share the load across them. For example, in this simple scenario we have a single POD serving a set of users. When the number of users increase, we deploy additional POD to balance the load across the two pods. If the demand further increases and If we were to run out of resources on the first node, we could deploy additional PODs across other nodes in the cluster. As you can see, the replication controller spans across multiple nodes in the cluster. It helps us balance the load across multiple pods on different nodes as well as scale our application when the demand increases.
Replication Controller and Replica set
◇ It is important to note that there are two similar terms. Replication Controller and Replica Set. Both have the same purpose, but they are not the same.

◇ Replication Controller is the older technology that is being replaced by Replica Set

◇ Replica set is the new recommended way to setup replication.

◇ However, whatever we discussed in the previous few slides remain applicable to both these technologies.

◇ There are minor differences in the way each works, and we will look at that in a bit

As such we will try to stick to Replica Sets in all our demos and implementations going forward. Let us now look at how we create a replication controller.
◇ As with the previous lecture, we start by creating a replication controller definition file.

◇ We will name it rcclefinition.ym I

◇ As with any kubernetes definition file, we will have 4 sections.

◇ The a piVersion,

◇ kind,

◇ metadata and

◇ spec

◇ The apiVersion is specific to what we are creating.

◇ In this case replication controller is supported in kubernetes apiVersion v1. So we will write it as

◇ The kind as we know is ReplicationController.

◇ Under metadata, we will add a name and we will call it rnyapp-rc. And we will also add a few labels, app and type and assign values to them

◇ The next is the most crucial part of the definition file and that is the specification written as spec

◇ For any kubernetes definition file, the spec section defines what is inside the object we are creating

c. In this case we know that the replication controller creates multiple instances of a POD
◇ But what POD? We create a template section under spec to provide a POD template to be used by the replication controller to create replicas

Now how do we DEFINE the POD template? It is not that hard because, we have already done that in the previous exercise. Remember, we created a pod-definition file in the previous exercise, We could re-use the contents of the same file to populate the template section. Move all the contents of the pod-definition file into the template section of the replication controller, except for the first two lines — which are apiVersion and kind.
Remember whatever we move must be UNDER the template section. Meaning, they should be intended to the right and have more spaces before them than the template line itself.
Looking at our file, we now have two metadata sections — one is for the Replication Controller and another for the POD and we have two spec sections one for each.
We have nested two definition files together. The replication controller being the parent and the pod-definition being the child
Now, there is something still missing. We haven't mentioned how many replicas we need in the replication controller
For that, add another property to the spec called replicas and input the number of replicas you need under it
Remember that the template and replicas are direct children of the spec section
So they are siblings and must be on the same vertical line : having equal number of spaces before them

Once the file is ready, run the kubectl create command and input the file using the —f parameter. The replication controller Is created. When the replication controller is created it first creates the PODs using the pod-definition template as many as required, which is 3 in this case
To view the list of created replication controllers run the kubectl get replication controller command and you will see the replication controller listed. We can also see the desired number of replicas or pods, the current number of replicas and how many of them are ready. If you would like to see the pods that were created by the replication controller, run the kubectl get pods command and you will see 3 pods running. Note that all of them are starting with the name of the replication controller which is myapp-rc indicating that they are all created automatically by the replication controller

Home

Kubernetes is a container management technology developed in Google lab to manage containerized applications in different kind of environments such as physical, virtual, and cloud infrastructure. It is an open source system which helps in creating and managing containerization of application. This tutorial provides an overview of different kind of features and functionalities of Kubernetes and teaches how to manage the containerized infrastructure and application deployment.

Audience

This tutorial has been prepared for those who want to understand the containerized infrastructure and deployment of application on containers. This tutorial will help in understanding the concepts of container management using Kubernetes.

Prerequisites

We assume anyone who wants to understand Kubernetes should have an understating of how the Docker works, how the Docker images are created, and how they work as a standalone unit. To reach to an advanced configuration in Kubernetes one should understand basic networking and how the protocol communication works.

Overview

Kubernetes in an open source container management tool hosted by Cloud Native Computing Foundation (CNCF).
~~This is also known as the enhanced version of Borg which was developed at Google to manage both long running processes and batch jobs, which was earlier handled by separate systems.~~
Kubernetes comes with a capability of automating deployment, scaling of application, and operations of application containers across clusters. Kubernetes is capable of creating container centric infrastructure.

Features of Kubernetes

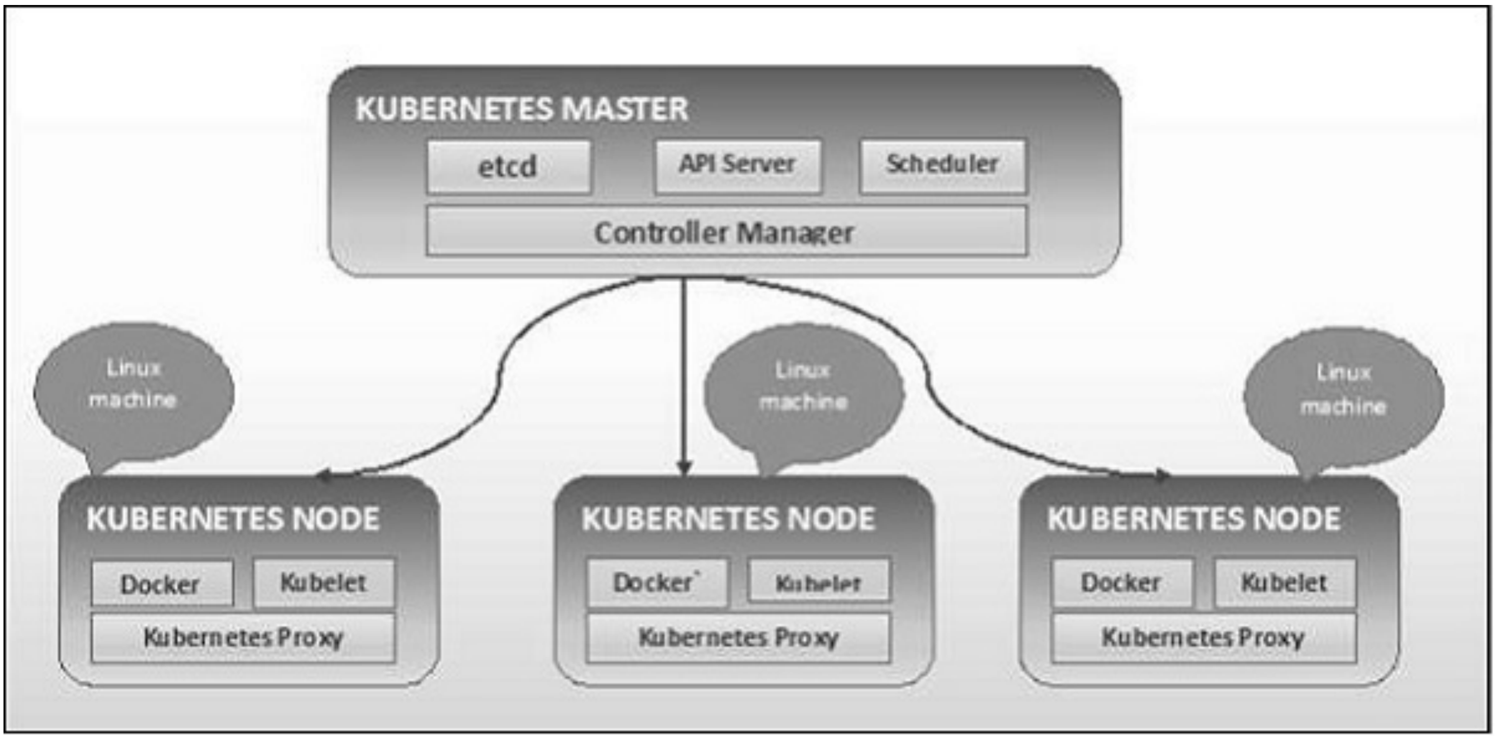
Following are some of the important features of Kubernetes.
• Continues development, integration and deployment
• Containerized infrastructure
• Application-centric management
• Auto-scalable infrastructure
• Environment consistency across development testing and production
• Loosely coupled infrastructure, where each component can act as a separate unit
• Higher density of resource utilization
• Predictable infrastructure which is going to be created

One of the key components of Kubernetes is, it can run application on clusters of physical and virtual machine infrastructure.
Kubernetes also has the capability to run applications on cloud.
Kubernetes **helps in moving from host-centric infrastructure to container-centric infrastructure.**

Architecture

Kubernetes - Cluster Architecture

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



The key components of master and node are defined in the following section.

Kubernetes - Master Machine Components

Following are the components of Kubernetes Master Machine.

1. etcd
2. API Server
3. Controller Manager

4. Scheduler
5.

etcd

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

Controller manager is responsible for most of the collectors that regulates the state of cluster and performs a task.
In general, Controller manager 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 controller manager runs different kind of controllers to handle nodes, endpoints, etc.
The key controllers are
1. replication controller, e
2. endpoint controller,
3. namespace controller, and
4. service account controller.

Scheduler

Scheduler 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

Following are the key components of Node server which are necessary to communicate with Kubernetes master.

1. Docker
2. Kubelet Service
3. Kubernetes Proxy Service
4.

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.

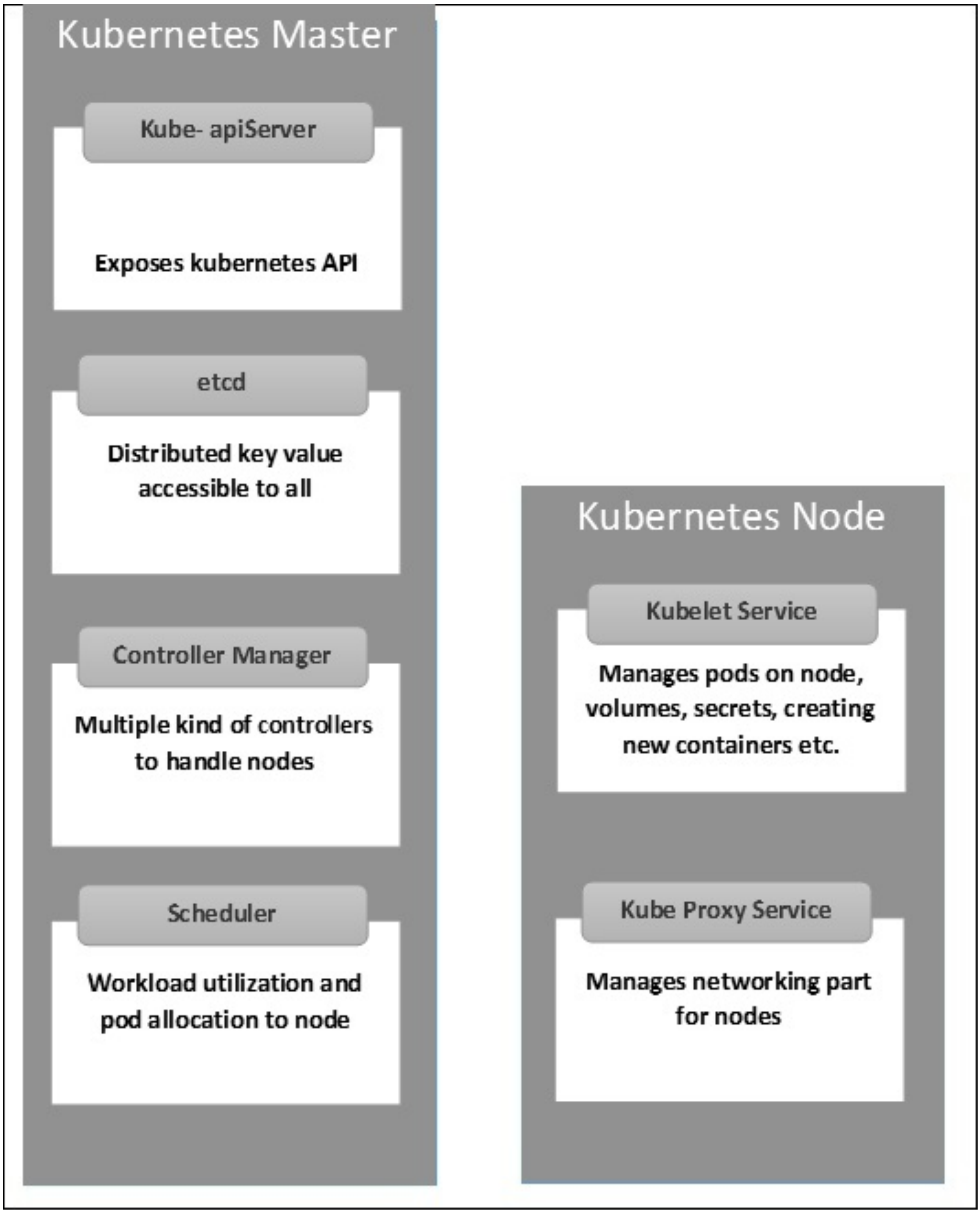
Kubernetes Proxy Service

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

The following illustrations show the structure of Kubernetes Master and Node.


Kubernetes Master
Kube- apiServer
Exposes kubernetes API
etcd
Distributed key value accessible to all
Controller Manager
Multiple kind of controllers to handle nodes
Scheduler
Workload utilization and pod allocation to node
Kubernetes Node
Kubelet Service
Manages pods on node, volumes, secrets, creating new containers etc.
Kube Proxy Service
Manages networking part for nodes

setup

It is important to set up the Virtual Datacenter (vDC) before setting up Kubernetes. This can be considered as a set of machines where they can communicate with each other via the network. For hands-on approach, you can set up vDC on **PROFITBRICKS** if you do not have a physical or cloud infrastructure set up.

Once the IaaS setup on any cloud is complete, you need to configure the **M-aster** and the **Node**.

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

Steps
1. Install Docker
2. Install Docker GPG Keys
3. Install Docker Engine
4. Install ETCD2.0
5. Install Kubernetes all all machines on the cluster
6. Confgure Kube API Server
7. Configure Kube Controller manager
8. Bring up Kube master
9. kubelet and kube-proxy configuration on Nodes
10. Start Nodes
11.

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.

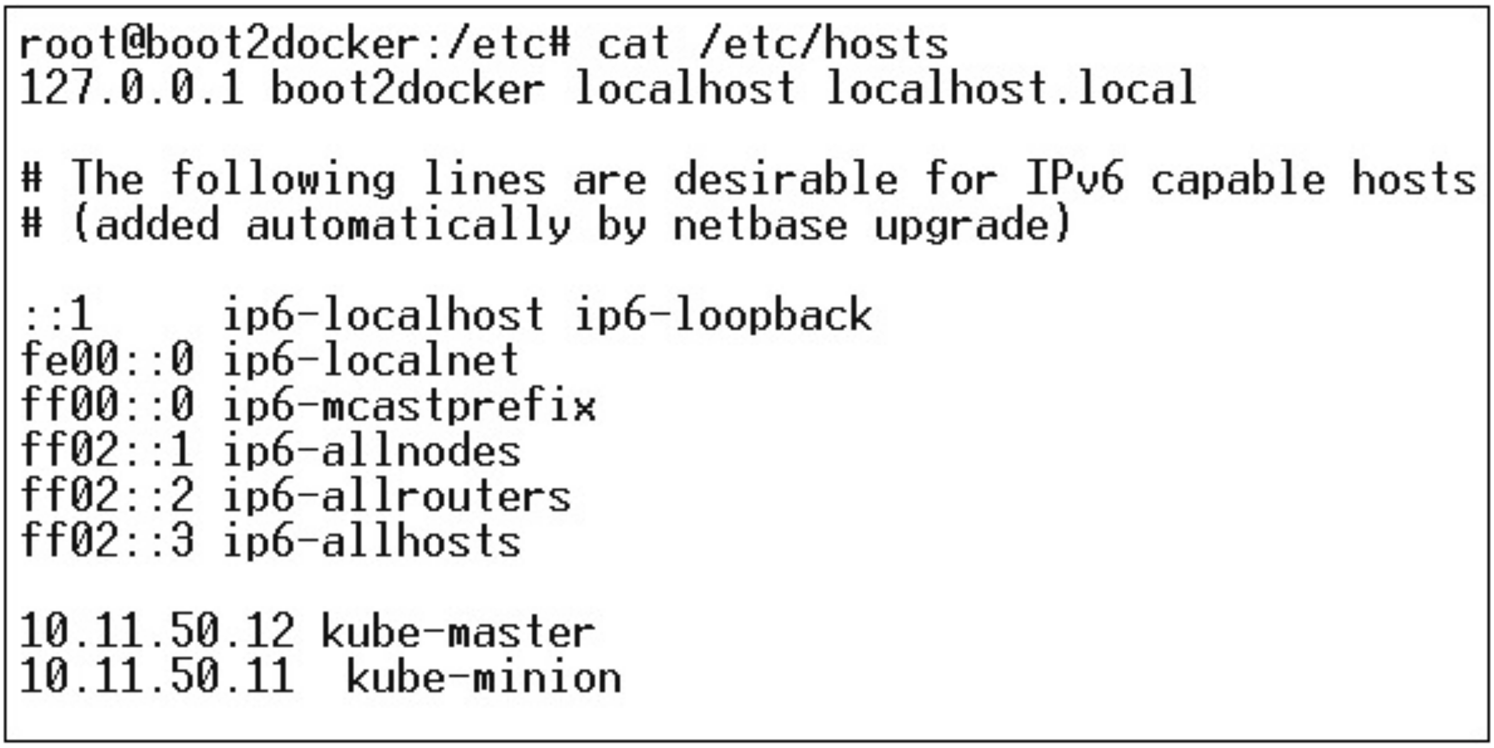
Now we are ready to build Kubernetes. We need to install Kubernetes on all the machines on the cluster.

```
$ 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
```

Images

Kubernetes (Docker) images are the key building blocks of Containerized Infrastructure.
As of now, we are only supporting Kubernetes to support Docker images.
Each container in a pod has its Docker image running inside it.

When we are configuring a pod, the image property in the configuration file has the same syntax as the Docker command does.
The configuration file has a field to define the image name, which we are planning to pull from the registry.

Following is the common configuration structure which will pull image from Docker registry and deploy in to Kubernetes container.

```
apiVersion: v1
kind: pod
metadata:
  name: Tesing_for_Image_pull -----------> 1
  spec:
    containers:
      - name: neo4j-server ------------------------> 2
      image: <Name of the Docker image>----------> 3
      imagePullPolicy: Always ------------->4
      command: ["echo", "SUCCESS"] ------------------->
```

In the above code, we have defined –

• **name: Tesing_for_Image_pull** – This name is given to identify and check what is the name of the container that would get created after pulling the images from Docker registry.

• **name: neo4j-server** – This is the name given to the container that we are trying to create. Like we have given neo4j-server.

• **image: <Name of the Docker image>** – This is the name of the image which we are trying to pull from the Docker or internal registry of images. We need to define a complete registry path along with the image name that we are trying to pull.

• **imagePullPolicy** − Always - This image pull policy defines that whenever we run this file to create the container, it will pull the same name again.

• **command: [“echo”, “SUCCESS”]** − With this, when we create the container and if everything goes fine, it will display a message when we will access the container.

In order to pull the image and create a container, we will run the following command.

```
$ kubectl create –f Tesing_for_Image_pull
```

Once we fetch the log, we will get the output as successful.

```
$ kubectl log Tesing_for_Image_pull
```

The above command will produce an output of success or we will get an output as failure.

Note − It is recommended that you try all the commands yourself.

Jobs

The main function of a job is to create one or more pod and tracks about the success of pods.
They ensure that the specified number of pods are completed successfully. When a specified number of successful run of pods is completed, then the job is considered complete.

Creating a Job

Use the following command to create a job –

```
apiVersion: v1
kind: Job ------------------------> 1
metadata:
  name: py
  spec:
  template:
    metadata
    name: py -------> 2
    spec:
      containers:
        - name: py ------------------------> 3
        image: python----------> 4
        command: ["python", "SUCCESS"]
        restartPocliy: Never --------> 5
```

In the above code, we have defined –

• **kind: Job →** We have defined the kind as Job which will tell **kubectl** that the **yaml** file being used is to create a job type pod.

• **Name:py →** This is the name of the template that we are using and the spec defines the template.

• **name: py →** we have given a name as **py** under container spec which helps to identify the Pod which is going to be created out of it.

• **Image: python →** the image which we are going to pull to create the container which will run inside the pod.

• **restartPolicy: Never →**This condition of image restart is given as never which means that if the container is killed or if it is false, then it will not restart itself.

We will create the job using the following command with yaml which is saved with the name **py.yaml**.

```
$ kubectl create –f py.yaml
```

The above command will create a job. If you want to check the status of a job, use the following command.

```
$ kubectl describe jobs/py
```

The above command will create a job. If you want to check the status of a job, use the following command.

Scheduled Job

Scheduled job in Kubernetes uses **Cronetes**, which takes Kubernetes job and launches them in Kubernetes cluster.

◇ Scheduling a job will run a pod at a specified point of time.

◇ A parodic job is created for it which invokes itself automatically.

Note – The feature of a scheduled job is supported by version 1.4 and the betch/v2alpha 1 API is turned on by passing the **–runtime-config=batch/v2alpha1** while bringing up the API server.

We will use the same yaml which we used to create the job and make it a scheduled job.

```
apiVersion: v1
kind: Job
metadata:
  name: py
spec:
  schedule: h/30 * * * * ? -------------------> 1
  template:
    metadata
      name: py
    spec:
      containers:
      - name: py
```

```
        image: python
        args:
/bin/sh -------> 2
-c
ps –eaf ------------> 3
restartPocliy: OnFailure
```

In the above code, we have defined –
◇ **schedule: h/30 * * * * ? →** To schedule the job to run in every 30 minutes.

◇ **/bin/sh:** This will enter in the container with /bin/sh

◇ **ps –eaf →** Will run ps -eaf command on the machine and list all the running process inside a container.

This scheduled job concept is useful when we are trying to build and run a set of tasks at a specified point of time and then complete the process.

Labelsandselectors

Labels
Labels are key-value pairs which are attached to pods, replication controller and services.
They are used as identifying attributes for objects such as pods and replication controller.
They can be added to an object at creation time and can be added or modified at the run time.

Selectors
Labels do not provide uniqueness.
In general, we can say many objects can carry the same labels.
Labels selector are core grouping primitive in Kubernetes. They are used by the users to select a set of objects.
Kubernetes API currently supports two type of selectors –
- Equality-based selectors
- Set-based selectors

Equality-based Selectors
They allow filtering by key and value. Matching objects should satisfy all the specified labels.

Set-based Selectors
Set-based selectors allow filtering of keys according to a set of values.

```
apiVersion: v1
kind: Service
metadata:
  name: sp-neo4j-standalone
spec:
  ports:
    - port: 7474
    name: neo4j
  type: NodePort
  selector:
    app: salesplatform ---------> 1
    component: neo4j -----------> 2
```

In the above code, we are using the label selector as **app: salesplatform** and component as **component: neo4j**.
Once we run the file using the **kubectl** command, it will create a service with the name **sp-neo4j-standalone** which will communicate on port 7474. The ype is **NodePort** with the new label selector as **app: salesplatform** and **co-mponent: neo4j**.

Namespaces

Namespace provides an additional qualification to a resource name.
This is helpful when multiple teams are using the same cluster and there is a potential of name collision.
It can be as a virtual wall between multiple clusters.

Functionality of Namespace

Following are some of the important functionalities of a Namespace in Kubernetes −

• Namespaces help pod-to-pod communication using the same namespace.
• Namespaces are virtual clusters that can sit on top of the same physical cluster.
• They provide logical separation between the teams and their environments.

1. Create a Namespace
2. Control the Namespace
3. Using Namespace in service

Create a Namespace

The following command is used to create a namespace.

```
apiVersion: v1
kind: Namespce
metadata
  name: elk
```

Control the Namespace

The following command is used to control the namespace.

```
$ kubectl create –f namespace.yml ---------> 1
$ kubectl get namespace -----------------> 2
$ kubectl get namespace <Namespace name> ------->3
$ kubectl describe namespace <Namespace name> ---->4
$ kubectl delete namespace <Namespace name>
```

In the above code,

◇ We are using the command to create a namespace.

◇ This will list all the available namespace.
◇ This will get a particular namespace whose name is specified in the command.
◇ This will describe the complete details about the service.
◇ This will delete a particular namespace present in the cluster.

Using Namespace in Service - Example

Following is an example of a sample file for using namespace in service.

```
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  namespace: elk
  labels:
    component: elasticsearch
spec:
  type: LoadBalancer
  selector:
    component: elasticsearch
  ports:
  - name: http
    port: 9200
    protocol: TCP
  - name: transport
    port: 9300
    protocol: TCP
```

In the above code, we are using the same namespace under service metadata with the name of **elk**.

Node

A node is a working machine in Kubernetes cluster which is also known as a minion.
They are working units which can be physical, VM, or a cloud instance.
Each node has all the required configuration required to run a pod on it such as the proxy service and kubelet service along with the Docker, which is used to run the Docker containers on the pod created on the node.
They are not created by Kubernetes but they are created externally either by the cloud service provider or the Kubernetes cluster manager on physical or VM machines.
The key component of Kubernetes to handle multiple nodes is the controller manager, which runs multiple kind of controllers to manage nodes. To manage nodes, Kubernetes creates an object of kind node which will validate that the object which is created is a valid node.

Service with Selector

```
apiVersion: v1
kind: node
metadata:
  name: < ip address of the node>
  labels:
    name: <lable name>
```

In JSON format the actual object is created which looks as follows −

```
{
  Kind: node
  apiVersion: v1
  "metadata":
  {
    "name": "10.01.1.10",
    "labels"
    {
      "name": "cluster 1 node"
    }
  }
}
```

Node Controller

They are the collection of services which run in the Kubernetes master and continuously monitor the node in the cluster on the basis of metadata.name.
If all the required services are running, then the node is validated and a newly created pod will be assigned to that node by the controller.
If it is not valid, then the master will not assign any pod to it and will wait until it becomes valid.

Kubernetes master registers the node automatically, if **–register-node** flag is true.

```
–register-node = true
```

However, if the cluster administrator wants to manage it manually then it could be done by turning the flat of –

```
–register-node = false
```

Service

A service can be defined as a logical set of pods.
It can be defined as an abstraction on the top of the pod which provides a single IP address and DNS name by which pods can be accessed.
With Service, it is very easy to manage load balancing configuration.
It helps pods to scale very easily.
A service is a REST object in Kubernetes whose definition can be posted to Kubernetes apiServer on the Kubernetes master to create a new instance.

Service without Selector

```
apiVersion: v1
kind: Service
metadata:
  name: Tutorial_point_service
spec:
  ports:
  - port: 8080
  targetPort: 31999
```

The above configuration will create a service with the name Tutorial_point_service.

Service Config File with Selector

```
apiVersion: v1
kind: Service
metadata:
  name: Tutorial_point_service
spec:
  selector:
    application: "My Application" -------------------> (Selector)
  ports:
  - port: 8080
  targetPort: 31999
```

In this example, we have a selector; so in order to transfer traffic, we need to create an endpoint manually.

```
apiVersion: v1
kind: Endpoints
metadata:
  name: Tutorial_point_service
```

```
subnets:
  address:
    "ip": "192.168.168.40" -------------------> (Selector)
  ports:
    - port: 8080
```

In the above code, we have created an endpoint which will route the traffic to the endpoint defined as “192.168.168.40:8080”.

Multi-Port Service Creation

```
apiVersion: v1
kind: Service
metadata:
  name: Tutorial_point_service
spec:
  selector:
    application: “My Application” -------------------> (Selector)
  ClusterIP: 10.3.0.12
  ports:
    -name: http
    protocol: TCP
    port: 80
    targetPort: 31999
  -name:https
    Protocol: TCP
    Port: 443
    targetPort: 31998
```

Types of Services

1. Cluster IP
2. NodePort
3. Load Balancer
4.

ClusterIP − This helps in restricting the service within the cluster. It exposes the service within the defined Kubernetes cluster.

```
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 31999
    name: NodeportService
```

NodePort – It will expose the service on a static port on the deployed node. A **ClusterIP** service, to which **NodePort** service will route, is automatically created. The service can be accessed from outside the cluster using the **N-odeIP:nodePort**.

```
spec:
  ports:
  - port: 8080
    nodePort: 31999
    name: NodeportService
    clusterIP: 10.20.30.40
```

Load Balancer – It uses cloud providers' load balancer. **NodePort** and **Clust-erIP** services are created automatically to which the external load balancer will route.

A full service **yaml** file with service type as Node Port. Try to create one yourself.

```
apiVersion: v1
kind: Service
metadata:
  name: appname
  labels:
    k8s-app: appname
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 31999
    name: omninginx
  selector:
    k8s-app: appname
    component: nginx
    env: env_name
```

Previous Page Print Pa

Pod

A pod is a collection of containers and its storage inside a node of a Kubernetes cluster.
It is possible to create a pod with multiple containers inside it.
For example, keeping a database container and data container in the same pod.

Types of Pod

There are two types of Pods –

- Single container pod
- Multi container pod

Single Container Pod

1. They can be simply created with the kubctl run command, where you have a defined image on the Docker registry which we will pull while creating a pod
2. This can also be done by creating the **yaml** file and then running the **kube-ctl create** command

Multi Container Pod

1. Multi container pods are created using **yaml mail** with the definition of the containers.

They can be simply created with the kubctl run command, where you have a defined image on the Docker registry which we will pull while creating a pod.

```
$ kubectl run <name of pod> --image=<name of the image from registry>
```

Example – We will create a pod with a tomcat image which is available on the Docker hub.

```
$ kubectl run tomcat --image = tomcat:8.0
```

This can also be done by creating the **yaml** file and then running the **kubectl create** command.

```
apiVersion: v1
kind: Pod
metadata:
  name: Tomcat
spec:
  containers:
  - name: Tomcat
   image: tomcat: 8.0
   ports:
containerPort: 7500
  imagePullPolicy: Always
```

Once the above **yaml** file is created, we will save the file with the name of **t-omcat.yml** and run the create command to run the document.

```
$ kubectl create –f tomcat.yml
```

It will create a pod with the name of tomcat. We can use the describe command along with **kubectl** to describe the pod.

Multi Container Pod

Multi container pods are created using **yaml mail** with the definition of the containers.

```
apiVersion: v1
kind: Pod
metadata:
  name: Tomcat
spec:
  containers:
  - name: Tomcat
   image: tomcat: 8.0
   ports:
containerPort: 7500
  imagePullPolicy: Always
  -name: Database
  Image: mongoDB
  Ports:
containerPort: 7501
  imagePullPolicy: Always
```

In the above code, we have created one pod with two containers inside it, one for tomcat and the other for MongoDB.

ReplicationController

Replication Controller is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle.
It is responsible for making sure that the specified number of pod replicas are running at any point of time.
It is used in time when one wants to make sure that the specified number of pod or at least one pod is running.
It has the capability to bring up or down the specified no of pod.
It is a best practice to use the replication controller to manage the pod life cycle rather than creating a pod again and again.

```
apiVersion: v1
kind: ReplicationController --------------------------> 1
metadata:
  name: Tomcat-ReplicationController --------------------------> 2
spec:
  replicas: 3 ------------------------> 3
  template:
    metadata:
      name: Tomcat-ReplicationController
    labels:
      app: App
      component: neo4j
    spec:
      containers:
      - name: Tomcat- ------------------------> 4
      image: tomcat: 8.0
      ports:
        - containerPort: 7474 ------------------------> 5
```

Setup Details

• **Kind: ReplicationController** → In the above code, we have defined the kind as replication controller which tells the **kubectl** that the **yaml** file is going to be used for creating the replication controller.

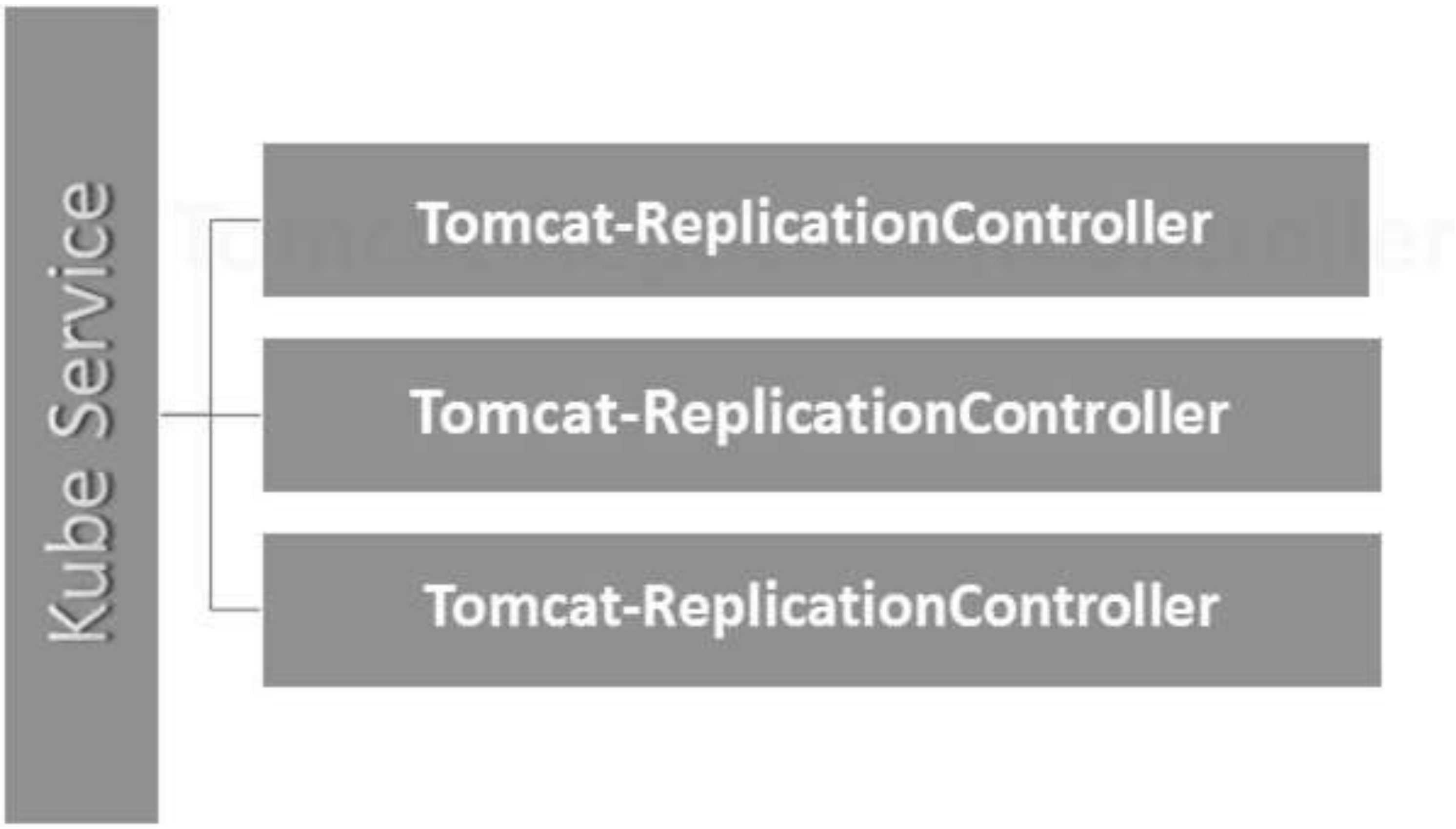
•

• **name: Tomcat-ReplicationController** → This helps in identifying the name with which the replication controller will be created. If we run the kubctl, get **rc < Tomcat-ReplicationController >** it will show the replication controller details.

• **replicas: 3** → This helps the replication controller to understand that it needs to maintain three replicas of a pod at any point of time in the pod lifecycle.

• **name: Tomcat** → In the spec section, we have defined the name as tomcat which will tell the replication controller that the container present inside the pods is tomcat.

• **containerPort: 7474** → It helps in making sure that all the nodes in the cluster where the pod is running the container inside the pod will be exposed on the same port 7474.


Kube Service
Tomcat-ReplicationController
Tomcat-ReplicationController
Tomcat-ReplicationController

Here, the Kubernetes service is working as a load balancer for three tomcat replicas.

Replica Sets

Replica Set ensures how many replica of pod should be running.
It can be considered as a replacement of replication controller.
The key difference between the replica set and the replication controller is, the replication controller only supports equality-based selector whereas the replica set supports set-based selector.

```
apiVersion: extensions/v1beta1 --------------------->1
kind: ReplicaSet --------------------------> 2
metadata:
  name: Tomcat-ReplicaSet
spec:
  replicas: 3
  selector:
    matchLables:
      tier: Backend ------------------> 3
    matchExpression:
{ key: tier, operation: In, values: [Backend]} --------------> 4
template:
  metadata:
    lables:
      app: Tomcat-ReplicaSet
      tier: Backend
    labels:
      app: App
      component: neo4j
  spec:
    containers:
    - name: Tomcat
    image: tomcat: 8.0
    ports:
    - containerPort: 7474
```

Setup Details

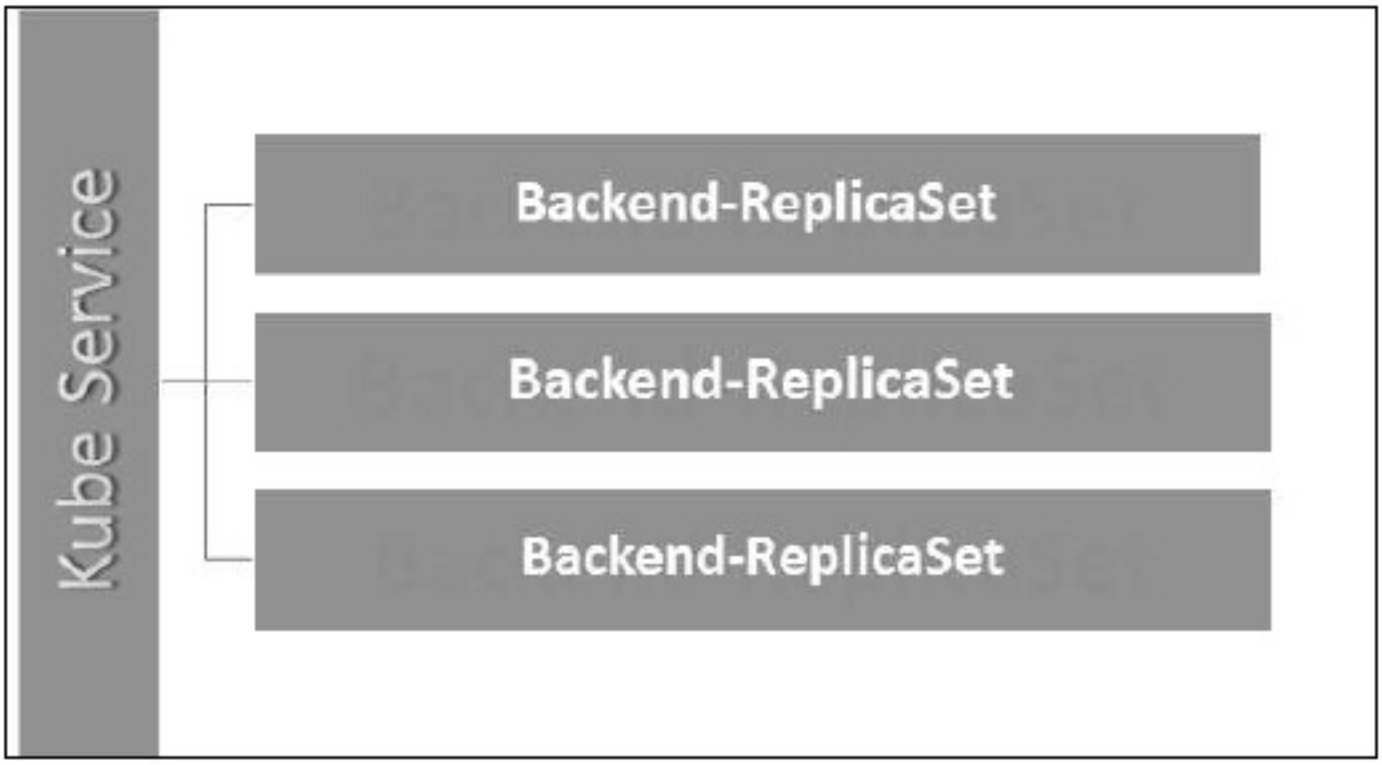
• **apiVersion: extensions/v1beta1** → In the above code, the API version is the advanced beta version of Kubernetes which supports the concept of replica set.

• **kind: ReplicaSet** → We have defined the kind as the replica set which helps kubectl to understand that the file is used to create a replica set.

• **tier: Backend** → We have defined the label tier as backend which creates a matching selector.

• **{key: tier, operation: In, values: [Backend]}** → This will help **matchExpression** to understand the matching condition we have defined and in the operation which is used by **matchlabel** to find details.

Run the above file using **kubectl** and create the backend replica set with the provided definition in the **yaml** file.



Deployments

Deployments are upgraded and higher version of replication controller. They manage the deployment of replica sets which is also an upgraded version of the replication controller.
They have the capability to update the replica set and are also capable of rolling back to the previous version.
They provide many updated features of **matchLabels** and **selectors**.
We have got a new controller in the Kubernetes master called the deployment controller which makes it happen.
It has the capability to change the deployment midway.

Changing the Deployment

Updating – The user can update the ongoing deployment before it is completed. In this, the existing deployment will be settled and new deployment will be created.
Deleting – The user can pause/cancel the deployment by deleting it before it is completed. Recreating the same deployment will resume it.
Rollback – We can roll back the deployment or the deployment in progress. The user can create or update the deployment by using **DeploymentSpec.PodTemplateSpec = oldRC.PodTemplateSpec.**

Deployment Strategies

Deployment strategies help in defining how the new RC should replace the existing RC.
Recreate – This feature will kill all the existing RC and then bring up the new ones. This results in quick deployment however it will result in downtime when the old pods are down and the new pods have not come up.
Rolling Update – This feature gradually brings down the old RC and brings up the new one. This results in slow deployment, however there is no deployment. At all times, few old pods and few new pods are available in this process.

The configuration file of Deployment looks like this.

```
apiVersion: extensions/v1beta1 --------------------->1
kind: Deployment --------------------------> 2
metadata:
  name: Tomcat-ReplicaSet
spec:
  replicas: 3
  template:
    metadata:
      lables:
        app: Tomcat-ReplicaSet
        tier: Backend
  spec:
    containers:
      - name: Tomcatimage:
        tomcat: 8.0
        ports:
          - containerPort: 7474
```

In the above code, the only thing which is different from the replica set is we have defined the kind as deployment.

Create Deployment

```
$ kubectl create –f Deployment.yaml -–record
deployment "Deployment" created Successfully.
```

Fetch the Deployment

```
$ kubectl get deployments
NAME        DESIRED    CURRENT    UP-TO-DATE    AVILABLE   AGE
Deployment      3        3        3           3      20s
```

Check the Status of Deployment

```
$ kubectl rollout status deployment/Deployment
```

Updating the Deployment

```
$ kubectl set image deployment/Deployment tomcat=tomcat:6.0
```

Rolling Back to Previous Deployment

```
$ kubectl rollout undo deployment/Deployment –to-revision=2
```

Volumes

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.

Types of Kubernetes Volume
Here is a list of some popular Kubernetes Volumes −

1. emptyDir
2. hostPath
3. gcePersistentDisk
4. awsElasticBlockStore
5. nfs
6. iscsi
7. flocker
8. glusterfs
9. rbd
10. cephfs
11. gitRepo
12. secret
13. persistentVolumeClaim
14. downwardAPI
15. azureDiskVolume

• **emptyDir** − It is a type of volume that is created when a Pod is first assigned to a Node. It remains active as long as the Pod is running on that node. The volume is initially empty and the containers in the pod can read and write the files in the emptyDir volume. Once the Pod is removed from the node, the data in the emptyDir is erased.

• **hostPath** – This type of volume mounts a file or directory from the host node's filesystem into your pod.

• **gcePersistentDisk** – This type of volume mounts a Google Compute Engine (GCE) Persistent Disk into your Pod. The data in a **gcePersistentDis-k** remains intact when the Pod is removed from the node.

• **awsElasticBlockStore** – This type of volume mounts an Amazon Web Services (AWS) Elastic Block Store into your Pod. Just like **gcePersistentD-isk**, the data in an **awsElasticBlockStore** remains intact when the Pod is removed from the node.

• **nfs** – An **nfs** volume allows an existing NFS (Network File System) to be mounted into your pod. The data in an **nfs** volume is not erased when the Pod is removed from the node. The volume is only unmounted.

• **iscsi** – An **iscsi** volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your pod.

• **flocker** – It is an open-source clustered container data volume manager. It is used for managing data volumes. A **flocker** volume allows a Flocker dataset to be mounted into a pod. If the dataset does not exist in Flocker, then you first need to create it by using the Flocker API.

• **glusterfs** – Glusterfs is an open-source networked filesystem. A glusterfs volume allows a glusterfs volume to be mounted into your pod.

• **rbd** – RBD stands for Rados Block Device. An **rbd** volume allows a Rados Block Device volume to be mounted into your pod. Data remains preserved after the Pod is removed from the node.

• **cephfs** – A **cephfs** volume allows an existing CephFS volume to be mounted into your pod. Data remains intact after the Pod is removed from the node.

• **gitRepo** – A **gitRepo** volume mounts an empty directory and clones a **git** r-epository into it for your pod to use.

• **secret** – A **secret** volume is used to pass sensitive information, such as passwords, to pods.

• **persistentVolumeClaim** – A **persistentVolumeClaim** volume is used to mount a PersistentVolume into a pod. PersistentVolumes are a way for users to "claim" durable storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.

• **downwardAPI** – A **downwardAPI** volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

• **azureDiskVolume** – An **AzureDiskVolume** is used to mount a Microsoft Azure Data Disk into a Pod.

Persistent Volume and Persistent Volume Claim

Persistent Volume (PV) – It's a piece of network storage that has been provisioned by the administrator.
It's a resource in the cluster which is independent of any individual pod that uses the PV.
Persistent Volume Claim (PVC) – The storage requested by Kubernetes for its pods is known as PVC.
The user does not need to know the underlying provisioning.
The claims must be created in the same namespace where the pod is created.

Creating Persistent Volume

```
kind: PersistentVolume ---------> 1
apiVersion: v1
metadata:
  name: pv0001 ------------------> 2
  labels:
    type: local
```

```
spec:
  capacity: -----------------------> 3
    storage: 10Gi ----------------------> 4
  accessModes:
    - ReadWriteOnce -------------------> 5
    hostPath:
      path: "/tmp/data01" -------------------------> 6
```

In the above code, we have defined −

◇ **kind: PersistentVolume** → We have defined the kind as PersistentVolume which tells kubernetes that the yaml file being used is to create the Persistent Volume.

◇ **name: pv0001** → Name of PersistentVolume that we are creating.

◇ **capacity:** → This spec will define the capacity of PV that we are trying to create.

◇ **storage: 10Gi** → This tells the underlying infrastructure that we are trying to claim 10Gi space on the defined path.

◇ **ReadWriteOnce** → This tells the access rights of the volume that we are creating.

◇ **path: "/tmp/data01"** → This definition tells the machine that we are trying to create volume under this path on the underlying infrastructure.

Creating PV

```
$ kubectl create –f local-01.yaml
persistentvolume "pv0001" created
```

Checking PV

```
$ kubectl get pv
NAME      CAPACITY    ACCESSMODES    STATUS     CLAIM    REASON    AGE
pv0001      10Gi        RWO       Available              14s
```

Describing PV

```
$ kubectl describe pv pv0001
```

Creating Persistent Volume Claim

```
kind: PersistentVolumeClaim --------------> 1
apiVersion: v1
metadata:
  name: myclaim-1 --------------------> 2
spec:
  accessModes:
    - ReadWriteOnce ------------------------> 3
  resources:
    requests:
      storage: 3Gi ---------------------> 4
```

In the above code, we have defined −

◇ **kind: PersistentVolumeClaim** → It instructs the underlying infrastructure that we are trying to claim a specified amount of space.

◇ **name: myclaim-1** → Name of the claim that we are trying to create.

◇ **ReadWriteOnce** → This specifies the mode of the claim that we are trying to create.

◇ **storage: 3Gi** → This will tell kubernetes about the amount of space we are trying to claim.

Creating PVC

```
$ kubectl create –f myclaim-1
persistentvolumeclaim "myclaim-1" created
```

Getting Details About PVC

```
$ kubectl get pvc
NAME      STATUS   VOLUME   CAPACITY   ACCESSMODES   AGE
myclaim-1   Bound    pv0001    10Gi       RWO       7s
```

Describe PVC

```
$ kubectl describe pv pv0001
```

Using PV and PVC with POD

```
kind: Pod
```

```
apiVersion: v1
metadata:
  name: mypod
  labels:
    name: frontendhttp
spec:
  containers:
  - name: myfrontend
    image: nginx
    ports:
    - containerPort: 80
      name: "http-server"
    volumeMounts: ----------------------------> 1
    - mountPath: "/usr/share/tomcat/html"
      name: mypd
  volumes: -----------------------> 2
    - name: mypd
      persistentVolumeClaim: ------------------------->3
      claimName: myclaim-1
```

In the above code, we have defined −

◇ **volumeMounts:** → This is the path in the container on which the mounting will take place.

◇ **Volume:** → This definition defines the volume definition that we are going to claim.

◇ **persistentVolumeClaim:** → Under this, we define the volume name which we are going to use in the defined pod.

Secrets

Secrets can be defined as Kubernetes objects used to store sensitive data such as user name and passwords with encryption.

There are multiple ways of creating secrets in Kubernetes.
1. Creating from txt files.
2. Creating from yaml file.

Creating From Text File

In order to create secrets from a text file such as user name and password, we first need to store them in a txt file and use the following command.

```
$ kubectl create secret generic tomcat-passwd –-from-file = ./username.txt –fromfile = ./.
password.txt
```

Creating From Yaml File

```
apiVersion: v1
kind: Secret
metadata:
name: tomcat-pass
type: Opaque
data:
  password: <User Password>
  username: <User Name>
```

Creating the Secret

```
$ kubectl create –f Secret.yaml
secrets/tomcat-pass
```

Using Secrets

Once we have created the secrets, it can be consumed in a pod or the replication controller as –

◇ Environment Variable
◇ Volume

As Environment Variable

In order to use the secret as environment variable, we will use **env** under the spec section of pod yaml file.

```
env:
- name: SECRET_USERNAME
  valueFrom:
    secretKeyRef:
      name: mysecret
      key: tomcat-pass
```

As Volume

```
spec:
  volumes:
    - name: "secretstest"
      secret:
        secretName: tomcat-pass
  containers:
    - image: tomcat:7.0
      name: awebserver
      volumeMounts:
        - mountPath: "/tmp/mysec"
        name: "secretstest"
```

Secret Configuration As Environment Variable

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: appname
spec:
replicas: replica_count
template:
  metadata:
    name: appname
  spec:
    nodeSelector:
      resource-group:
    containers:
      - name: appname
        image:
        imagePullPolicy: Always
        ports:
        - containerPort: 3000
        env: -----------------------------> 1
    - name: ENV
      valueFrom:
        configMapKeyRef:
          name: appname
          key: tomcat-secrets
```

In the above code, under the **env** definition, we are using secrets as environment variable in the replication controller.

Secrets As Volume Mount

```
apiVersion: v1
kind: pod
metadata:
  name: appname
spec:
  metadata:
    name: appname
  spec:
  volumes:
    - name: "secretstest"
      secret:
        secretName: tomcat-pass
  containers:
    - image: tomcat: 8.0
      name: awebserver
      volumeMounts:
        - mountPath: "/tmp/mysec"
        name: "secretstest"
```

Network Policy

Network Policy defines how the pods in the same namespace will communicate with each other and the network endpoint.
It requires **extensions/v1beta1/networkpolicies** to be enabled in the runtime configuration in the API server.
Its resources use labels to select the pods and define rules to allow traffic to a specific pod in addition to which is defined in the namespace.

First, we need to configure Namespace Isolation Policy. Basically, this kind of networking policies are required on the load balancers.

```
kind: Namespace
apiVersion: v1
metadata:
  annotations:
    net.beta.kubernetes.io/network-policy: |
    {
      "ingress":
      {
        "isolation": "DefaultDeny"
      }
    }
```

```
$ kubectl annotate ns <namespace> "net.beta.kubernetes.io/network-policy =
{\"ingress\": {\"isolation\": \"DefaultDeny\"}}"
```

Once the namespace is created, we need to create the Network Policy.

Network Policy Yaml

```
kind: NetworkPolicy
apiVersion: extensions/v1beta1
metadata:
  name: allow-frontend
  namespace: myns
spec:
  podSelector:
    matchLabels:
```

```
    role: backend
ingress:
- from:
  - podSelector:
    matchLabels:
      role: frontend
ports:
  - protocol: TCP
    port: 6379
```

Advanced

API

Kubernetes API serves as a foundation for declarative configuration schema for the system.
Kubectl command-line tool can be used to create, update, delete, and get API object.
Kubernetes API acts a communicator among different components of Kubernetes.

Adding API to Kubernetes

Adding a new API to Kubernetes will add new features to Kubernetes, which will increase the functionality of Kubernetes.
However, alongside it will also increase the cost and maintainability of the system.
In order to create a balance between the cost and complexity, there are a few sets defined for it.
The API which is getting added should be useful to more than 50% of the users.
There is no other way to implement the functionality in Kubernetes.
Exceptional circumstances are discussed in the community meeting of Kubernetes, and then API is added.

API Changes

In order to increase the capability of Kubernetes, changes are continuously introduced to the system.
It is done by Kubernetes team to add the functionality to Kubernetes without removing or impacting the existing functionality of the system.
To demonstrate the general process, here is an (hypothetical) example −

- A user POSTs a Pod object to **/api/v7beta1/...**

- The JSON is unmarshalled into a **v7beta1.Pod** structure

- Default values are applied to the **v7beta1.Pod**

- The **v7beta1.Pod** is converted to an **api.Pod** structure

• The **api.Pod** is validated, and any errors are returned to the user

• The **api.Pod** is converted to a v6.Pod (because v6 is the latest stable version)

• The **v6.Pod** is marshalled into JSON and written to **etcd**

Now that we have the Pod object stored, a user can GET that object in any supported API version. For example –
◇ A user GETs the Pod from **/api/v5/...**

◇ The JSON is read from **etcd** and **unmarshalled** into a **v6.Pod** structure

◇ Default values are applied to the **v6.Pod**

◇ The **v6.Pod** is converted to an api.Pod structure

◇ The **api.Pod** is converted to a **v5.Pod** structure

◇ The **v5.Pod** is marshalled into JSON and sent to the user

The implication of this process is that API changes must be done carefully and backward compatibly.

API Versioning

To make it easier to support multiple structures, Kubernetes supports multiple API versions each at different API path such as **/api/v1** or **/apsi/extensions/v1beta1**
Versioning standards at Kubernetes are defined in multiple standards.

Alpha Level
◇ This version contains alpha (e.g. v1alpha1)
◇ This version may be buggy; the enabled version may have bugs
◇ Support for bugs can be dropped at any point of time.
◇ Recommended to be used in short term testing only as the support may not be present all the time.

Beta Level
◇ The version name contains beta (e.g. v2beta3)
◇ The code is fully tested and the enabled version is supposed to be stable.
◇ The support of the feature will not be dropped; there may be some small changes.
◇ Recommended for only non-business-critical uses because of the potential for incompatible changes in subsequent releases.

Stable Level
◇ The version name is **vX** where **X** is an integer.
◇ Stable versions of features will appear in the released software for many subsequent versions.

Kubectl

Kubectl is the command line utility to interact with Kubernetes API.
It is an interface which is used to communicate and manage pods in Kubernetes cluster.
One needs to set up kubectl to local in order to interact with Kubernetes cluster.

Setting Kubectl

Download the executable to the local workstation using the curl command.

On Linux

```
$ curl -O https://storage.googleapis.com/kubernetesrelease/
release/v1.5.2/bin/linux/amd64/kubectl
```

On OS X workstation

```
$ curl -O https://storage.googleapis.com/kubernetesrelease/
release/v1.5.2/bin/darwin/amd64/kubectl
```

After download is complete, move the binaries in the path of the system.

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

Configuring Kubectl

Following are the steps to perform the configuration operation.

```
$ kubectl config set-cluster default-cluster --server = https://${MASTER_HOST} --
certificate-authority = ${CA_CERT}

$ kubectl config set-credentials default-admin --certificateauthority = ${
CA_CERT} --client-key = ${ADMIN_KEY} --clientcertificate = ${
ADMIN_CERT}

$ kubectl config set-context default-system --cluster = default-cluster --
user = default-admin
$ kubectl config use-context default-system
```

• Replace **${MASTER_HOST}** with the master node address or name used in the previous steps.

• Replace **${CA_CERT}** with the absolute path to the **ca.pem** created in the previous steps.

• Replace **${ADMIN_KEY}** with the absolute path to the **admin-key.pem** created in the previous steps.

• Replace **${ADMIN_CERT}** with the absolute path to the **admin.pem** created in the previous steps.

Verifying the Setup

To verify if the **kubectl** is working fine or not, check if the Kubernetes client is set up correctly.

```
$ kubectl get nodes

NAME      LABELS                                   STATUS
Vipin.com  Kubernetes.io/hostname = vipin.mishra.com   Ready
```

Kubectl commands

Kubectl controls the Kubernetes Cluster.
It is one of the key components of Kubernetes which runs on the workstation on any machine when the setup is done.
It has the capability to manage the nodes in the cluster.

Kubectl commands are used to interact and manage Kubernetes objects and the cluster.
In this chapter, we will discuss a few commands used in Kubernetes via kubectl.

kubectl annotate − It updates the annotation on a resource.

```
$kubectl annotate [--overwrite] (-f FILENAME | TYPE NAME) KEY_1=VAL_1 ...
KEY_N = VAL_N [--resource-version = version]
```

For example,

```
kubectl annotate pods tomcat description = 'my frontend'
```

kubectl api-versions − It prints the supported versions of API on the cluster.

```
$ kubectl api-version;
```

kubectl apply − It has the capability to configure a resource by file or stdin.

```
$ kubectl apply –f <filename>
```

kubectl attach − This attaches things to the running container.

```
$ kubectl attach <pod> –c <container>
$ kubectl attach 123456-7890 -c tomcat-conatiner
```

kubectl autoscale − This is used to auto scale pods which are defined such as Deployment, replica set, Replication Controller.

```
$ kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min = MINPODS] --
max = MAXPODS [--cpu-percent = CPU] [flags]
$ kubectl autoscale deployment foo --min = 2 --max = 10
```

kubectl cluster-info − It displays the cluster Info.

```
$ kubectl cluster-info
```

kubectl cluster-info dump − It dumps relevant information regarding cluster for debugging and diagnosis.

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

kubectl config − Modifies the kubeconfig file.

```
$ kubectl config <SUBCOMMAD>
```

$ kubectl config –-kubeconfig <String of File name>

kubectl config current-context – It displays the current context.

$ kubectl config current-context
#deploys the current context

kubectl config delete-cluster – Deletes the specified cluster from kubeconfig.

$ kubectl config delete-cluster <Cluster Name>

kubectl config delete-context – Deletes a specified context from kubeconfig.

$ kubectl config delete-context <Context Name>

kubectl config get-clusters – Displays cluster defined in the kubeconfig.

$ kubectl config get-cluster
$ kubectl config get-cluster <Cluser Name>

kubectl config get-contexts – Describes one or many contexts.

$ kubectl config get-context <Context Name>

kubectl config set-cluster – Sets the cluster entry in Kubernetes.

$ kubectl config set-cluster NAME [--server = server] [--certificateauthority = path/to/certificate/authority] [--insecure-skip-tls-verify = true]

kubectl config set-context – Sets a context entry in kubernetes entrypoint.

$ kubectl config set-context NAME [--cluster = cluster_nickname] [--user = user_nickname] [--namespace = namespace]
$ kubectl config set-context prod –user = vipin-mishra

kubectl config set-credentials – Sets a user entry in kubeconfig.

$ kubectl config set-credentials cluster-admin --username = vipin --password = uXFGweU9l35qcif

kubectl config set – Sets an individual value in kubeconfig file.

$ kubectl config set PROPERTY_NAME PROPERTY_VALUE

kubectl config unset – It unsets a specific component in kubectl.

$ kubectl config unset PROPERTY_NAME PROPERTY_VALUE

kubectl config use-context – Sets the current context in kubectl file.

$ kubectl config use-context <Context Name>

kubectl config view

$ kubectl config view
$ kubectl config view –o jsonpath='{.users[?(@.name == "e2e")].user.password}'

kubectl cp – Copy files and directories to and from containers.

$ kubectl cp <Files from source> <Files to Destinatiion>
$ kubectl cp /tmp/foo <some-pod>:/tmp/bar -c <specific-container>

kubectl create – To create resource by filename of or stdin. To do this, JSON or YAML formats are accepted.

$ kubectl create –f <File Name>

```
$ cat <file name> | kubectl create –f -
```

In the same way, we can create multiple things as listed using the **create** c-ommand along with **kubectl**.

• deployment
• namespace
• quota
• secret docker-registry
• secret
• secret generic
• secret tls
• serviceaccount
• service clusterip
• service loadbalancer
• service nodeport

kubectl delete – Deletes resources by file name, stdin, resource and names.

```
$ kubectl delete –f ([-f FILENAME] | TYPE [(NAME | -l label | --all)])
```

kubectl describe – Describes any particular resource in kubernetes. Shows details of resource or a group of resources.

```
$ kubectl describe <type> <type name>
$ kubectl describe pod tomcat
```

kubectl drain – This is used to drain a node for maintenance purpose. It prepares the node for maintenance. This will mark the node as unavailable so that it should not be assigned with a new container which will be created.

```
$ kubectl drain tomcat –force
```

kubectl edit – It is used to end the resources on the server. This allows to directly edit a resource which one can receive via the command line tool.

```
$ kubectl edit <Resource/Name | File Name)
Ex.
$ kubectl edit rc/tomcat
```

kubectl exec – This helps to execute a command in the container.

```
$ kubectl exec POD <-c CONTAINER > -- COMMAND < args...>
$ kubectl exec tomcat 123-5-456 date
```

kubectl expose – This is used to expose the Kubernetes objects such as pod, replication controller, and service as a new Kubernetes service. This has the capability to expose it via a running container or from a **yaml** file.

```
$ kubectl expose (-f FILENAME | TYPE NAME) [--port=port] [--protocol = TCP|UDP]
[--target-port = number-or-name] [--name = name] [--external-ip = external-ip-ofservice]
```

```
[--type = type]
$ kubectl expose rc tomcat –-port=80 –target-port = 30000
$ kubectl expose –f tomcat.yaml –port = 80 –target-port =
```

kubectl get − This command is capable of fetching data on the cluster about the Kubernetes resources.

```
$ kubectl get [(-o|--output=)json|yaml|wide|custom-columns=...|custom-columnsfile=...|
go-template=...|go-template-file=...|jsonpath=...|jsonpath-file=...]
(TYPE [NAME | -l label] | TYPE/NAME ...) [flags]
```

For example,

```
$ kubectl get pod <pod name>
$ kubectl get service <Service name>
```

kubectl logs − They are used to get the logs of the container in a pod. Printing the logs can be defining the container name in the pod. If the POD has only one container there is no need to define its name.

```
$ kubectl logs [-f] [-p] POD [-c CONTAINER]
```

Example

```
$ kubectl logs tomcat.
$ kubectl logs –p –c tomcat.8
```

kubectl port-forward − They are used to forward one or more local port to pods.

```
$ kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT
[...[LOCAL_PORT_N:]REMOTE_PORT_N]
$ kubectl port-forward tomcat 3000 4000
$ kubectl port-forward tomcat 3000:5000
```

kubectl replace − Capable of replacing a resource by file name or **stdin**.

```
$ kubectl replace -f FILENAME
$ kubectl replace –f tomcat.yml
$ cat tomcat.yml | kubectl replace –f -
```

kubectl rolling-update − Performs a rolling update on a replication controller. Replaces the specified replication controller with a new replication controller by updating a POD at a time.

```
$ kubectl rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] --
image = NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC)
$ kubectl rolling-update frontend-v1 –f freontend-v2.yaml
```

kubectl rollout − It is capable of managing the rollout of deployment.

```
$ Kubectl rollout <Sub Command>
$ kubectl rollout undo deployment/tomcat
```

Apart from the above, we can perform multiple tasks using the rollout such

as –
◇ rollout history
◇ rollout pause
◇ rollout resume
◇ rollout status
◇ rollout undo

kubectl run – Run command has the capability to run an image on the Kubernetes cluster.
```
$ kubectl run NAME --image = image [--env = "key = value"] [--port = port] [--
replicas = replicas] [--dry-run = bool] [--overrides = inline-json] [--command] --
[COMMAND] [args...]
$ kubectl run tomcat --image = tomcat:7.0
$ kubectl run tomcat –-image = tomcat:7.0 –port = 5000
```
kubectl scale – It will scale the size of Kubernetes Deployments, ReplicaSet, Replication Controller, or job.
```
$ kubectl scale [--resource-version = version] [--current-replicas = count] --
replicas = COUNT (-f FILENAME | TYPE NAME )
$ kubectl scale –-replica = 3 rs/tomcat
$ kubectl scale –replica = 3 tomcat.yaml
```
kubectl set image – It updates the image of a pod template.
```
$ kubectl set image (-f FILENAME | TYPE NAME)
CONTAINER_NAME_1 = CONTAINER_IMAGE_1 ... CONTAINER_NAME_N =
CONTAINER_IMAGE_N
$ kubectl set image deployment/tomcat busybox = busybox ngnix = ngnix:1.9.1
$ kubectl set image deployments, rc tomcat = tomcat6.0 --all
```
kubectl set resources – It is used to set the content of the resource. It updates resource/limits on object with pod template.
```
$ kubectl set resources (-f FILENAME | TYPE NAME) ([--limits = LIMITS & --
requests = REQUESTS]
$ kubectl set resources deployment tomcat -c = tomcat --
limits = cpu = 200m,memory = 512Mi
```
kubectl top node – It displays CPU/Memory/Storage usage. The top command allows you to see the resource consumption for nodes.
```
$ kubectl top node [node Name]
```
The same command can be used with a pod as well.

creating a app

In order to create an application for Kubernetes deployment, we need to first create the application on the Docker.
This can be done in two ways −

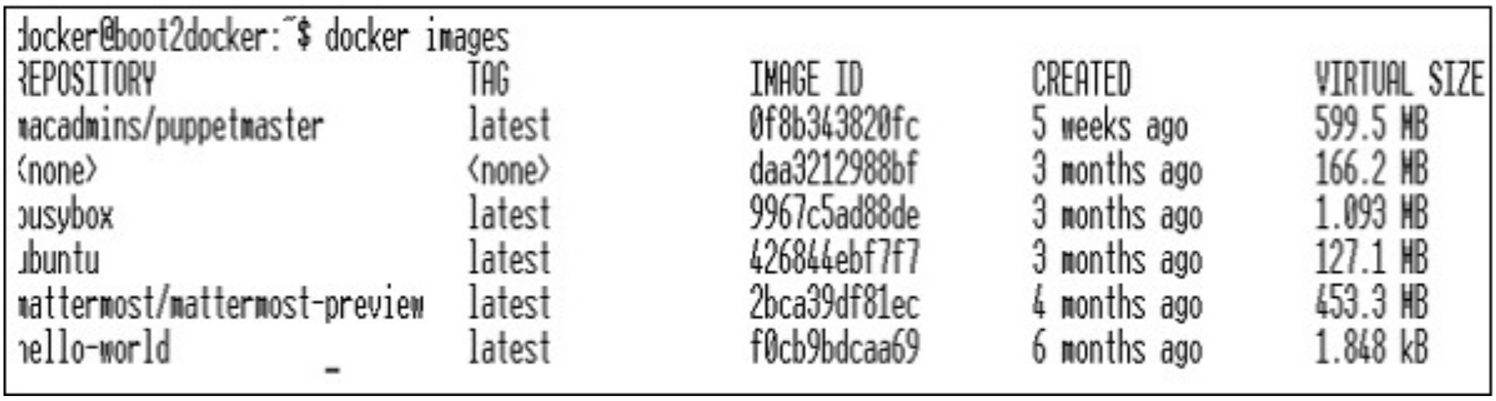
• By downloading
• From Docker file

By Downloading

The existing image can be downloaded from Docker hub and can be stored on the local Docker registry.
In order to do that, run the Docker **pull** command.

```
$ docker pull --help
Usage: docker pull [OPTIONS] NAME[:TAG|@DIGEST]
Pull an image or a repository from the registry
  -a, --all-tags = false    Download all tagged images in the repository
  --help = false            Print usage
```

Following will be the output of the above code.



```
docker@boot2docker:~$ docker images
REPOSITORY                       TAG        IMAGE ID        CREATED         VIRTUAL SIZE
macadmins/puppetmaster           latest     0f8b343820fc    5 weeks ago     599.5 MB
<none>                           <none>     daa3212988bf    3 months ago    166.2 MB
busybox                          latest     9967c5ad88de    3 months ago    1.093 MB
ubuntu                           latest     426844ebf7f7    3 months ago    127.1 MB
mattermost/mattermost-preview    latest     2bca39df81ec    4 months ago    453.3 MB
hello-world                      latest     f0cb9bdcaa69    6 months ago    1.848 kB
```

The above screenshot shows a set of images which are stored in our local Docker registry.
If we want to build a container from the image which consists of an application to test, we can do it using the Docker run command.

```
$ docker run –i –t unbunt /bin/bash
```

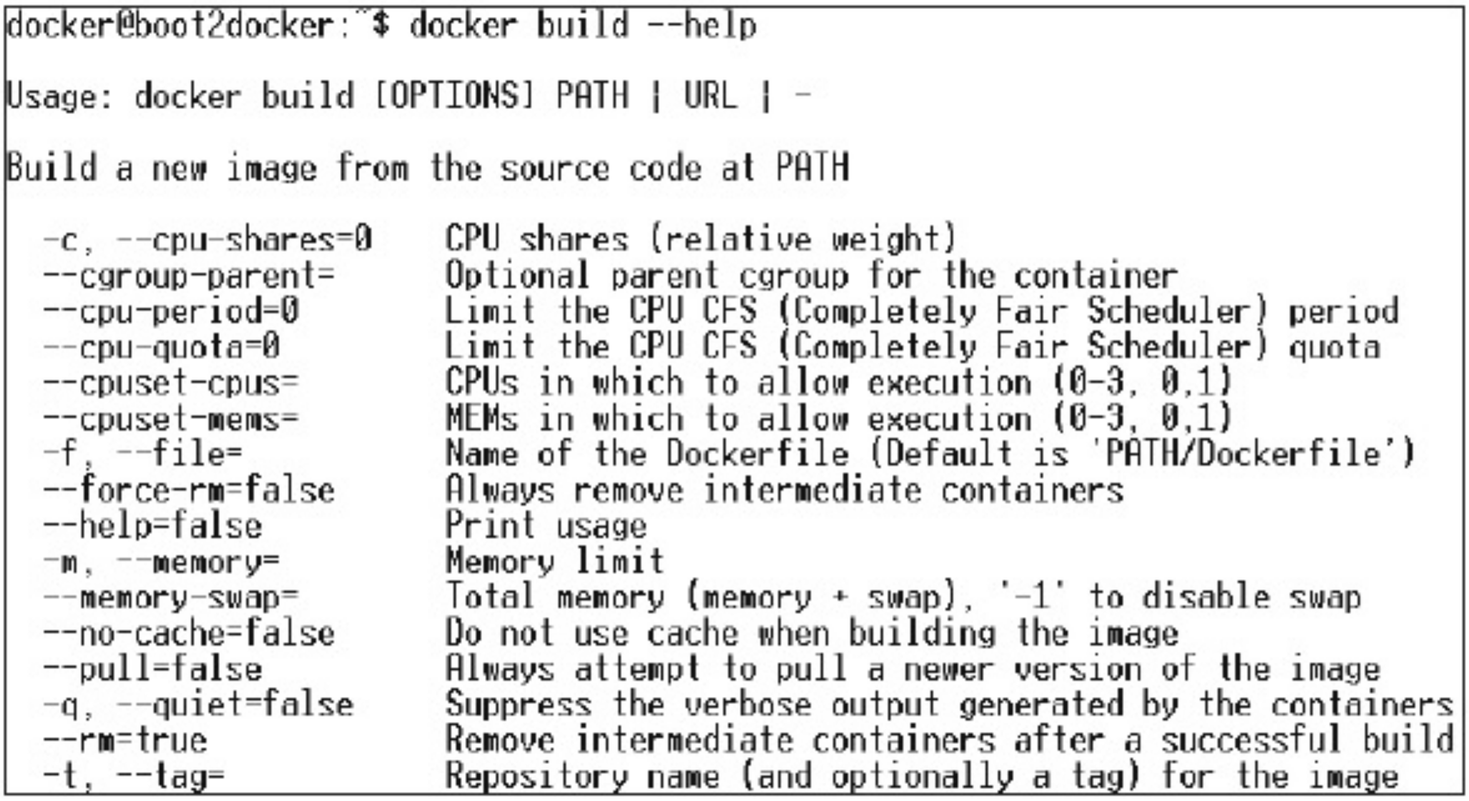
From Docker File

In order to create an application from the Docker file, we need to first create a Docker file.

Following is an example of Jenkins Docker file.

```
FROM ubuntu:14.04
MAINTAINER vipinkumarmishra@virtusapolaris.com
ENV REFRESHED_AT 2017-01-15
RUN apt-get update -qq && apt-get install -qqy curl
RUN curl https://get.docker.io/gpg | apt-key add -
RUN echo deb http://get.docker.io/ubuntu docker main > /etc/apt/↩
sources.list.d/docker.list
RUN apt-get update -qq && apt-get install -qqy iptables ca-↩
certificates lxc openjdk-6-jdk git-core lxc-docker
ENV JENKINS_HOME /opt/jenkins/data
ENV JENKINS_MIRROR http://mirrors.jenkins-ci.org
RUN mkdir -p $JENKINS_HOME/plugins
RUN curl -sf -o /opt/jenkins/jenkins.war -L $JENKINS_MIRROR/war-↩
stable/latest/jenkins.war
RUN for plugin in chucknorris greenballs scm-api git-client git ↩
ws-cleanup ;\
do curl -sf -o $JENKINS_HOME/plugins/${plugin}.hpi \
-L $JENKINS_MIRROR/plugins/${plugin}/latest/${plugin}.hpi ↩
; done
ADD ./dockerjenkins.sh /usr/local/bin/dockerjenkins.sh
RUN chmod +x /usr/local/bin/dockerjenkins.sh
VOLUME /var/lib/docker
EXPOSE 8080
ENTRYPOINT [ "/usr/local/bin/dockerjenkins.sh" ]
```

Once the above file is created, save it with the name of Dockerfile and cd to the file path. Then, run the following command.


```
docker@boot2docker:~$ docker build --help

Usage: docker build [OPTIONS] PATH | URL | -

Build a new image from the source code at PATH

  -c, --cpu-shares=0     CPU shares (relative weight)
  --cgroup-parent=       Optional parent cgroup for the container
  --cpu-period=0         Limit the CPU CFS (Completely Fair Scheduler) period
  --cpu-quota=0          Limit the CPU CFS (Completely Fair Scheduler) quota
  --cpuset-cpus=         CPUs in which to allow execution (0-3, 0,1)
  --cpuset-mems=         MEMs in which to allow execution (0-3, 0,1)
  -f, --file=            Name of the Dockerfile (Default is 'PATH/Dockerfile')
  --force-rm=false       Always remove intermediate containers
  --help=false           Print usage
  -m, --memory=          Memory limit
  --memory-swap=         Total memory (memory + swap), '-1' to disable swap
  --no-cache=false       Do not use cache when building the image
  --pull=false           Always attempt to pull a newer version of the image
  -q, --quiet=false      Suppress the verbose output generated by the containers
  --rm=true              Remove intermediate containers after a successful build
  -t, --tag=             Repository name (and optionally a tag) for the image
```

$ sudo docker build -t jamtur01/Jenkins .

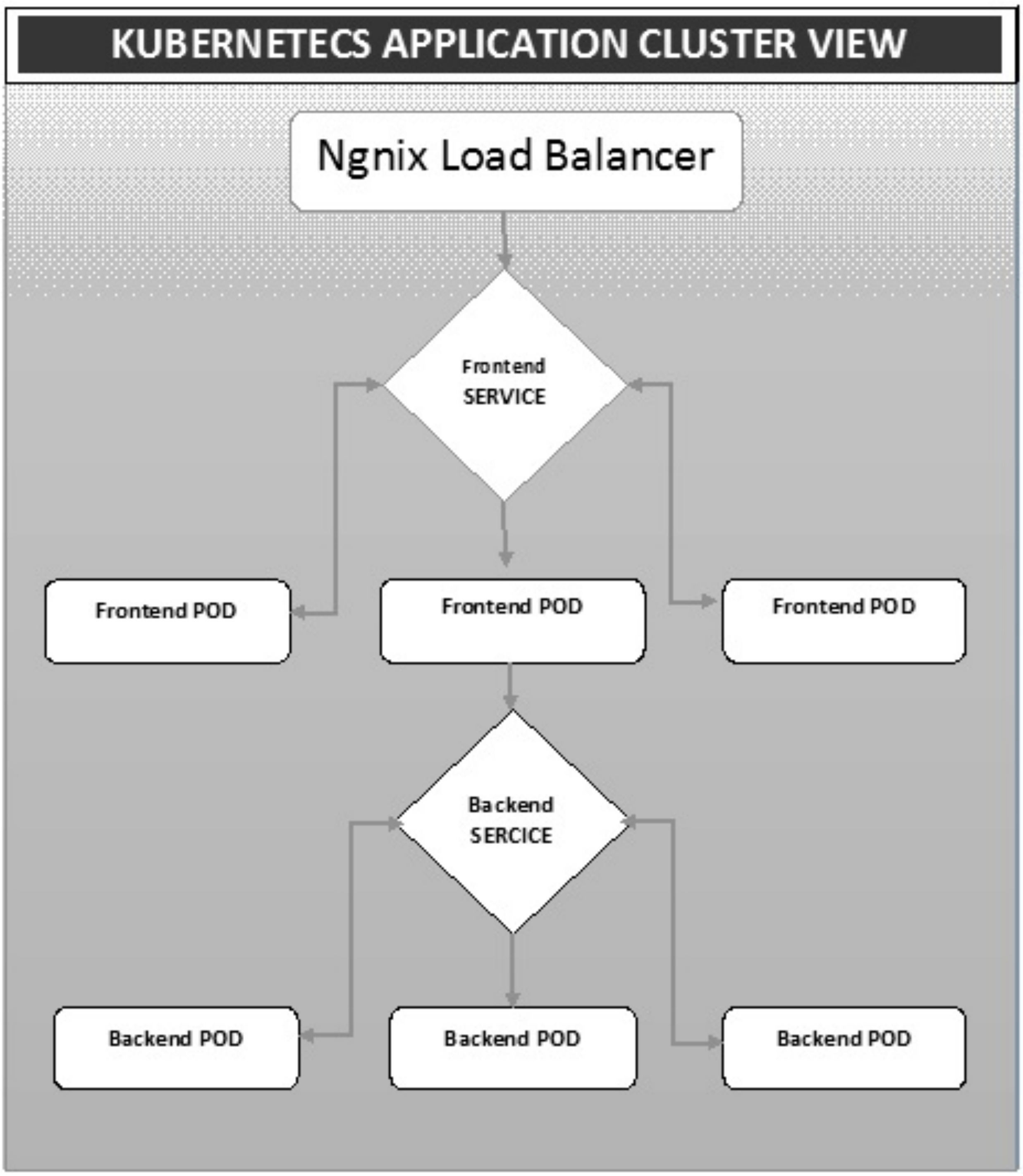
Once the image is built, we can test if the image is working fine and can be converted to a container.

$ docker run –i –t jamtur01/Jenkins /bin/bash

App Deployment

Deployment is a method of converting images to containers and then allocating those images to pods in the Kubernetes cluster.
This also helps in setting up the application cluster which includes deployment of service, pod, replication controller and replica set.
The cluster can be set up in such a way that the applications deployed on the pod can communicate with each other.

In this setup, we can have a load balancer setting on top of one application diverting traffic to a set of pods and later they communicate to backend pods.
The communication between pods happen via the service object built in Kubernetes.



Ngnix Load Balancer Yaml File

```
apiVersion: v1
kind: Service
metadata:
  name: oppv-dev-nginx
    labels:
      k8s-app: omni-ppv-api
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 31999
    name: omninginx
  selector:
    k8s-app: appname
    component: nginx
    env: dev
```

Ngnix Replication Controller Yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: appname
spec:
  replicas: replica_count
  template:
    metadata:
      name: appname
      labels:
        k8s-app: appname
        component: nginx
          env: env_name
spec:
  nodeSelector:
    resource-group: oppv
  containers:
    - name: appname
    image: IMAGE_TEMPLATE
    imagePullPolicy: Always
    ports:
      - containerPort: 8080
      resources:
        requests:
          memory: "request_mem"
          cpu: "request_cpu"
        limits:
          memory: "limit_mem"
          cpu: "limit_cpu"
        env:
        - name: BACKEND_HOST
          value: oppv-env_name-node:3000
```

Frontend Service Yaml File

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: appname
  labels:
    k8s-app: appname
spec:
  type: NodePort
  ports:
  - name: http
    port: 3000
    protocol: TCP
    targetPort: 3000
  selector:
    k8s-app: appname
    component: nodejs
    env: dev
```

Frontend Replication Controller Yaml File

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: Frontend
spec:
  replicas: 3
  template:
    metadata:
      name: frontend
      labels:
        k8s-app: Frontend
        component: nodejs
        env: Dev
spec:
  nodeSelector:
    resource-group: oppv
  containers:
    - name: appname
      image: IMAGE_TEMPLATE
      imagePullPolicy: Always
      ports:
        - containerPort: 3000
        resources:
          requests:
            memory: "request_mem"
            cpu: "limit_cpu"
            limits:
            memory: "limit_mem"
            cpu: "limit_cpu"
        env:
          - name: ENV
          valueFrom:
          configMapKeyRef:
          name: appname
          key: config-env
```

Backend Service Yaml File

```
apiVersion: v1
kind: Service
metadata:
  name: backend
  labels:
    k8s-app: backend
spec:
  type: NodePort
  ports:
  - name: http
    port: 9010
    protocol: TCP
    targetPort: 9000
  selector:
    k8s-app: appname
    component: play
    env: dev
```

Backed Replication Controller Yaml File

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: backend
spec:
  replicas: 3
  template:
    metadata:
      name: backend
    labels:
      k8s-app: beckend
      component: play
      env: dev
spec:
  nodeSelector:
    resource-group: oppv
    containers:
      - name: appname
        image: IMAGE_TEMPLATE
        imagePullPolicy: Always
        ports:
        - containerPort: 9000
        command: [ "./docker-entrypoint.sh" ]
        resources:
          requests:
            memory: "request_mem"
            cpu: "request_cpu"
          limits:
            memory: "limit_mem"
            cpu: "limit_cpu"
        volumeMounts:
          - name: config-volume
          mountPath: /app/vipin/play/conf
      volumes:
        - name: config-volume
        configMap:
        name: appname
```

AutoScaling

Autoscaling is one of the key features in Kubernetes cluster.
It is a feature in which the cluster is capable of increasing the number of nodes as the demand for service response increases and decrease the number of nodes as the requirement decreases.
This feature of auto scaling is currently supported in Google Cloud Engine (GCE) and Google Container Engine (GKE) and will start with AWS pretty soon.
In order to set up scalable infrastructure in GCE, we need to first have an active GCE project with features of Google cloud monitoring, google cloud logging, and stackdriver enabled.
First, we will set up the cluster with few nodes running in it. Once done, we need to set up the following environment variable.

Environment Variable

```
export NUM_NODES = 2
export KUBE_AUTOSCALER_MIN_NODES = 2
export KUBE_AUTOSCALER_MAX_NODES = 5
export KUBE_ENABLE_CLUSTER_AUTOSCALER = true
```

Once done, we will start the cluster by running **kube-up.sh**. This will create a cluster together with cluster auto-scalar add on.

```
./cluster/kube-up.sh
```

On creation of the cluster, we can check our cluster using the following kubectl command.

```
$ kubectl get nodes
NAME                     STATUS                  AGE
kubernetes-master             Ready,SchedulingDisabled    10m
kubernetes-minion-group-de5q    Ready                10m
kubernetes-minion-group-yhdx    Ready                8m
```

Now, we can deploy an application on the cluster and then enable the horizontal pod autoscaler. This can be done using the following command.

```
$ kubectl autoscale deployment <Application Name> --cpu-percent = 50 --min = 1 --
max = 10
```

The above command shows that we will maintain at least one and maximum 10 replica of the POD as the load on the application increases.
We can check the status of autoscaler by running the **$kubclt get hpa** com-

mand. We will increase the load on the pods using the following command.

```
$ kubectl run -i --tty load-generator --image = busybox /bin/sh
$ while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done
```

We can check the **hpa** by running **$ kubectl get hpa** command.

```
$ kubectl get hpa
NAME       REFERENCE                  TARGET CURRENT
php-apache   Deployment/php-apache/scale   50%   310%

MINPODS  MAXPODS   AGE
  1      20     2m

$ kubectl get deployment php-apache
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache     7        7        7         3      4m
```

We can check the number of pods running using the following command.

```
jsz@jsz-desk2:~/k8s-src$ kubectl get pods
php-apache-2046965998-3ewo6 0/1      Pending 0       1m
php-apache-2046965998-8m03k 1/1      Running 0       1m
php-apache-2046965998-ddpgp 1/1      Running 0       5m
php-apache-2046965998-lrik6 1/1      Running 0       1m
php-apache-2046965998-nj465 0/1      Pending 0       1m
php-apache-2046965998-tmwg1 1/1      Running 0       1m
php-apache-2046965998-xkbw1 0/1      Pending 0       1m
```

And finally, we can get the node status.

```
$ kubectl get nodes
NAME                      STATUS                  AGE
kubernetes-master              Ready,SchedulingDisabled     9m
kubernetes-minion-group-6z5i    Ready                  43s
kubernetes-minion-group-de5q    Ready                  9m
kubernetes-minion-group-yhdx    Ready                  9m
```

Dashboard Setup

Setting up Kubernetes dashboard involves several steps with a set of tools required as the prerequisites to set it up.

- Docker (1.3+)
- go (1.5+)
- nodejs (4.2.2+)
- npm (1.3+)
- java (7+)
- gulp (3.9+)
- Kubernetes (1.1.2+)

Setting Up the Dashboard

```
$ sudo apt-get update && sudo apt-get upgrade
```

Installing Python

```
$ sudo apt-get install python
$ sudo apt-get install python3
```

Installing GCC

```
$ sudo apt-get install gcc-4.8 g++-4.8
```

Installing make

```
$ sudo apt-get install make
```

Installing Java

```
$ sudo apt-get install openjdk-7-jdk
```

Installing Node.js

```
$ wget https://nodejs.org/dist/v4.2.2/node-v4.2.2.tar.gz
$ tar -xzf node-v4.2.2.tar.gz
$ cd node-v4.2.2
$ ./configure
$ make
$ sudo make install
```

Installing gulp

```
$ npm install -g gulp
$ npm install gulp
```

Verifying Versions

Java Version

```
$ java –version
java version "1.7.0_91"
OpenJDK Runtime Environment (IcedTea 2.6.3) (7u91-2.6.3-1~deb8u1+rpi1)
OpenJDK Zero VM (build 24.91-b01, mixed mode)

$ node –v
V4.2.2

$ npn -v
2.14.7

$ gulp -v
[09:51:28] CLI version 3.9.0

$ sudo gcc --version
gcc (Raspbian 4.8.4-1) 4.8.4
Copyright (C) 2013 Free Software Foundation, Inc. This is free software;
see the source for copying conditions. There is NO warranty; not even for
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Installing GO

```
$ git clone https://go.googlesource.com/go
$ cd go
$ git checkout go1.4.3
$ cd src
```

Building GO

```
$ ./all.bash
$ vi /root/.bashrc
```

In the .bashrc

```
  export GOROOT = $HOME/go
  export PATH = $PATH:$GOROOT/bin

$ go version
go version go1.4.3 linux/arm
```

Installing Kubernetes Dashboard

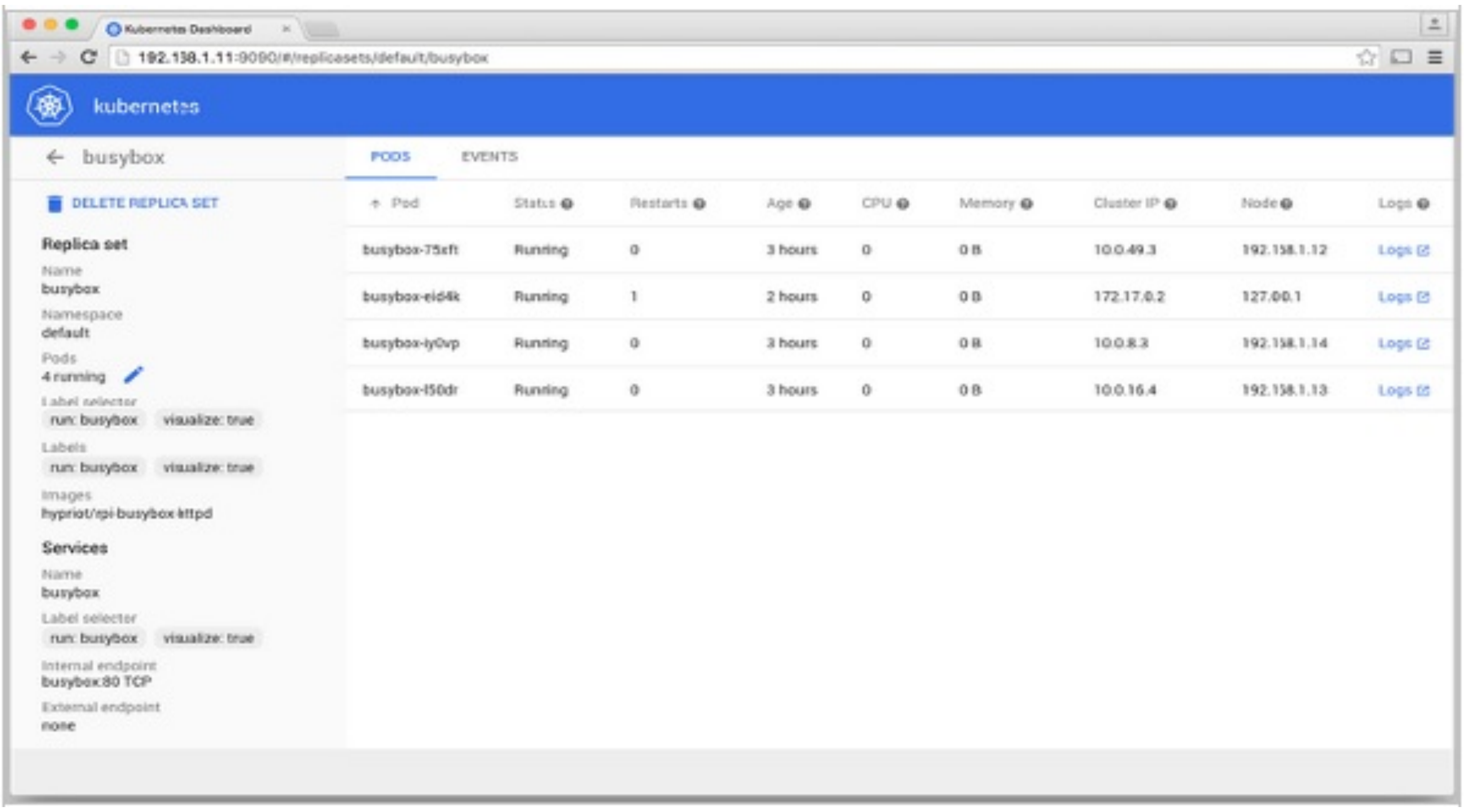
```
$ git clone https://github.com/kubernetes/dashboard.git
$ cd dashboard
$ npm install -g bower
```

Running the Dashboard

```
$ git clone https://github.com/kubernetes/dashboard.git
$ cd dashboard
$ npm install -g bower
$ gulp serve
[11:19:12] Requiring external module babel-core/register
[11:20:50] Using gulpfile ~/dashboard/gulpfile.babel.js
[11:20:50] Starting 'package-backend-source'...
[11:20:50] Starting 'kill-backend'...
[11:20:50] Finished 'kill-backend' after 1.39 ms
[11:20:50] Starting 'scripts'...
[11:20:53] Starting 'styles'...
[11:21:41] Finished 'scripts' after 50 s
[11:21:42] Finished 'package-backend-source' after 52 s
[11:21:42] Starting 'backend'...
[11:21:43] Finished 'styles' after 49 s
[11:21:43] Starting 'index'...
[11:21:44] Finished 'index' after 1.43 s
[11:21:44] Starting 'watch'...
[11:21:45] Finished 'watch' after 1.41 s
[11:23:27] Finished 'backend' after 1.73 min
[11:23:27] Starting 'spawn-backend'...
[11:23:27] Finished 'spawn-backend' after 88 ms
[11:23:27] Starting 'serve'...
2016/02/01 11:23:27 Starting HTTP server on port 9091
2016/02/01 11:23:27 Creating API client for
2016/02/01 11:23:27 Creating Heapster REST client for http://localhost:8082
[11:23:27] Finished 'serve' after 312 ms
[BS] [BrowserSync SPA] Running...
[BS] Access URLs:
--------------------------------------
Local: http://localhost:9090/
External: http://192.168.1.21:9090/
--------------------------------------
UI: http://localhost:3001
UI External: http://192.168.1.21:3001
--------------------------------------
[BS] Serving files from: /root/dashboard/.tmp/serve
[BS] Serving files from: /root/dashboard/src/app/frontend
```

```
[BS] Serving files from: /root/dashboard/src/app
```

The Kubernetes Dashboard



Replica set

Name
busybox

Namespace
default

Pods
4 running

Label selector
run: busybox visualize: true

Labels
run: busybox visualize: true

Images
hypriot/rpi-busybox-httpd

Services

Name
busybox

Label selector
run: busybox visualize: true

Internal endpoint
busybox:80 TCP

External endpoint
none

Pod	Status	Restarts	Age	CPU	Memory	Cluster IP	Node	Logs
busybox-75xft	Running	0	3 hours	0	0 B	10.0.49.3	192.158.1.12	Logs
busybox-eid4k	Running	1	2 hours	0	0 B	172.17.0.2	127.0.0.1	Logs
busybox-iy0vp	Running	0	3 hours	0	0 B	10.0.8.3	192.158.1.14	Logs
busybox-l50dr	Running	0	3 hours	0	0 B	10.0.16.4	192.158.1.13	Logs

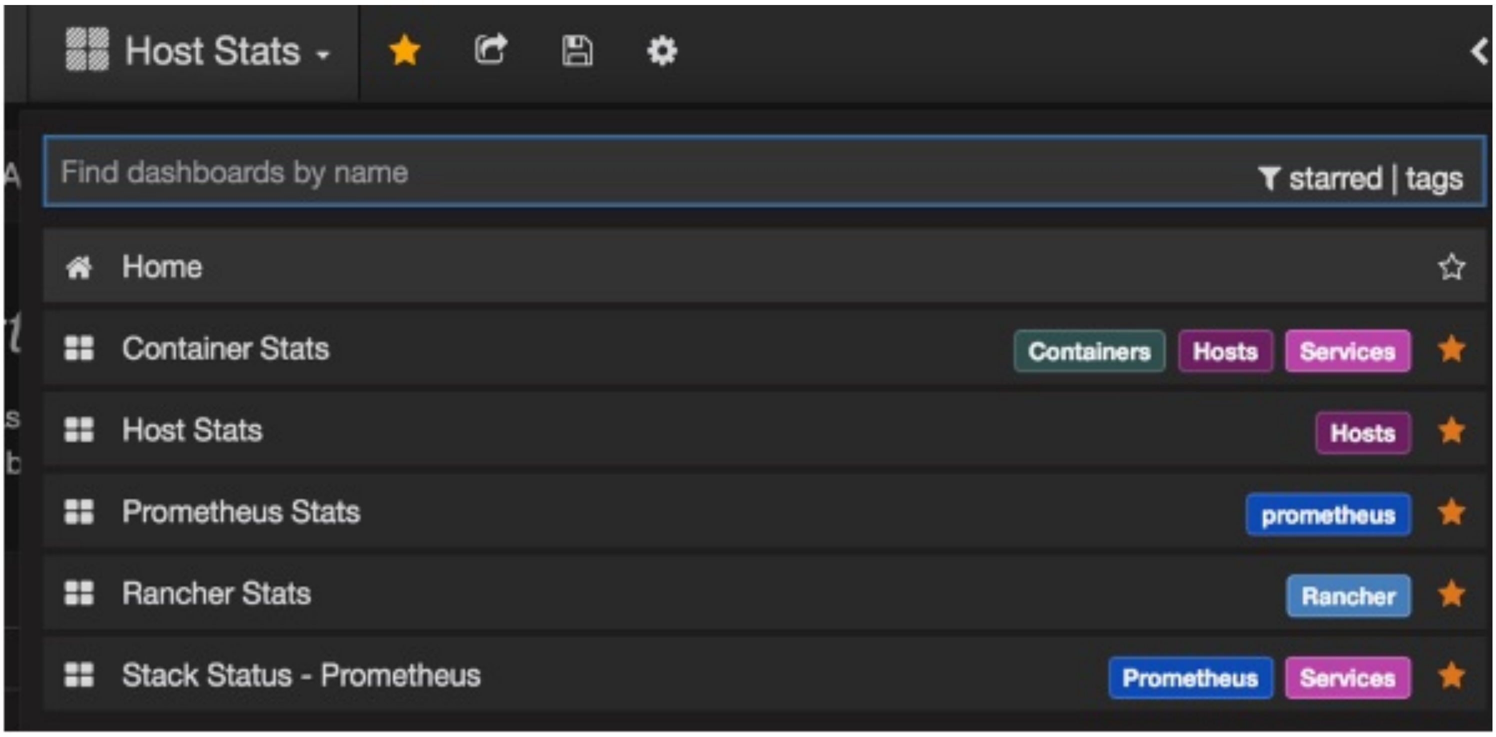
Monitoring

Monitoring is one of the key component for managing large clusters. For this, we have a number of tools.

Monitoring with Prometheus

It is a monitoring and alerting system.
It was built at SoundCloud and was open sourced in 2012.
It handles the multi-dimensional data very well.
Prometheus has multiple components to participate in monitoring –

- **Prometheus** – It is the core component that scraps and stores data.
- **Prometheus node explore** – Gets the host level matrices and exposes them to Prometheus.
- **Ranch-eye** – is an **haproxy** and exposes **cAdvisor** stats to Prometheus.
- **Grafana** – Visualization of data.
- **InfuxDB** – Time series database specifically used to store data from rancher.
- **Prom-ranch-exporter** – It is a simple node.js application, which helps in querying Rancher server for the status of stack of service.



Home	
Container Stats	Containers Hosts Services
Host Stats	Hosts
Prometheus Stats	prometheus
Rancher Stats	Rancher
Stack Status - Prometheus	Prometheus Services

Sematext Docker Agent

It is a modern Docker-aware metrics, events, and log collection agent.
It runs as a tiny container on every Docker host and collects logs, metrics, and events for all cluster node and containers.
It discovers all containers (one pod might contain multiple containers) including containers for Kubernetes core services, if the core services are deployed in Docker containers. After its deployment, all logs and metrics are immediately available out of the box.

Deploying Agents to Nodes

Kubernetes provides DeamonSets which ensures pods are added to the cluster.

Configuring SemaText Docker Agent

It is configured via environment variables.
◇ Get a free account at apps.sematext.com, if you don't have one already.
◇ Create an SPM App of type "Docker" to obtain the SPM App Token. SPM App will hold your Kubernetes performance metrics and event.
◇ Create a Logsene App to obtain the Logsene App Token. Logsene App will hold your Kubernetes logs.
◇ Edit values of LOGSENE_TOKEN and SPM_TOKEN in the DaemonSet definition as shown below.

■ Grab the latest sematext-agent-daemonset.yml (raw plain-text) template (also shown below).
■ Store it somewhere on the disk.
■ Replace the SPM_TOKEN and LOGSENE_TOKEN placeholders with your SPM and Logsene App tokens.

Create DaemonSet Object

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: sematext-agent
spec:
  template:
    metadata:
      labels:
        app: sematext-agent
    spec:
      selector: {}
      dnsPolicy: "ClusterFirst"
      restartPolicy: "Always"
      containers:
      - name: sematext-agent
        image: sematext/sematext-agent-docker:latest
        imagePullPolicy: "Always"
        env:
        - name: SPM_TOKEN
          value: "REPLACE THIS WITH YOUR SPM TOKEN"
        - name: LOGSENE_TOKEN
          value: "REPLACE THIS WITH YOUR LOGSENE TOKEN"
        - name: KUBERNETES
          value: "1"
        volumeMounts:
          - mountPath: /var/run/docker.sock
            name: docker-sock
          - mountPath: /etc/localtime
            name: localtime
        volumes:
          - name: docker-sock
            hostPath:
              path: /var/run/docker.sock
          - name: localtime
            hostPath:
              path: /etc/localtime
```

Running the Sematext Agent Docker with kubectl

```
$ kubectl create -f sematext-agent-daemonset.yml
daemonset "sematext-agent-daemonset" created
```

Kubernetes Log

Kubernetes containers' logs are not much different from Docker container logs.

However, Kubernetes users need to view logs for the deployed pods.

Hence, it is very useful to have Kubernetes-specific information available for log search, such as –

◇ Kubernetes namespace
◇ Kubernetes pod name
◇ Kubernetes container name
◇ Docker image name
◇ Kubernetes UID

Using ELK Stack and LogSpout

ELK stack includes Elasticsearch, Logstash, and Kibana. To collect and forward the logs to the logging platform, we will use LogSpout (though there are other options such as FluentD).
The following code shows how to set up ELK cluster on Kubernetes and create service for ElasticSearch –

```
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  namespace: elk
  labels:
    component: elasticsearch
spec:
  type: LoadBalancer
  selector:
    component: elasticsearch
  ports:
  - name: http
    port: 9200
    protocol: TCP
  - name: transport
    port: 9300
    protocol: TCP
```

Creating Replication Controller

```
apiVersion: v1
```

```
kind: ReplicationController
metadata:
  name: es
  namespace: elk
  labels:
    component: elasticsearch
spec:
  replicas: 1
  template:
    metadata:
      labels:
        component: elasticsearch
spec:
serviceAccount: elasticsearch
containers:
  - name: es
    securityContext:
    capabilities:
    add:
    - IPC_LOCK
  image: quay.io/pires/docker-elasticsearch-kubernetes:1.7.1-4
  env:
  - name: KUBERNETES_CA_CERTIFICATE_FILE
  value: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  - name: NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
  - name: "CLUSTER_NAME"
    value: "myesdb"
  - name: "DISCOVERY_SERVICE"
    value: "elasticsearch"
  - name: NODE_MASTER
    value: "true"
  - name: NODE_DATA
    value: "true"
  - name: HTTP_ENABLE
    value: "true"
ports:
- containerPort: 9200
  name: http
  protocol: TCP
```

```
- containerPort: 9300
volumeMounts:
- mountPath: /data
  name: storage
volumes:
  - name: storage
    emptyDir: {}
```

Kibana URL

For Kibana, we provide the Elasticsearch URL as an environment variable.

```
- name: KIBANA_ES_URL
value: "http://elasticsearch.elk.svc.cluster.local:9200"
- name: KUBERNETES_TRUST_CERT
value: "true"
```

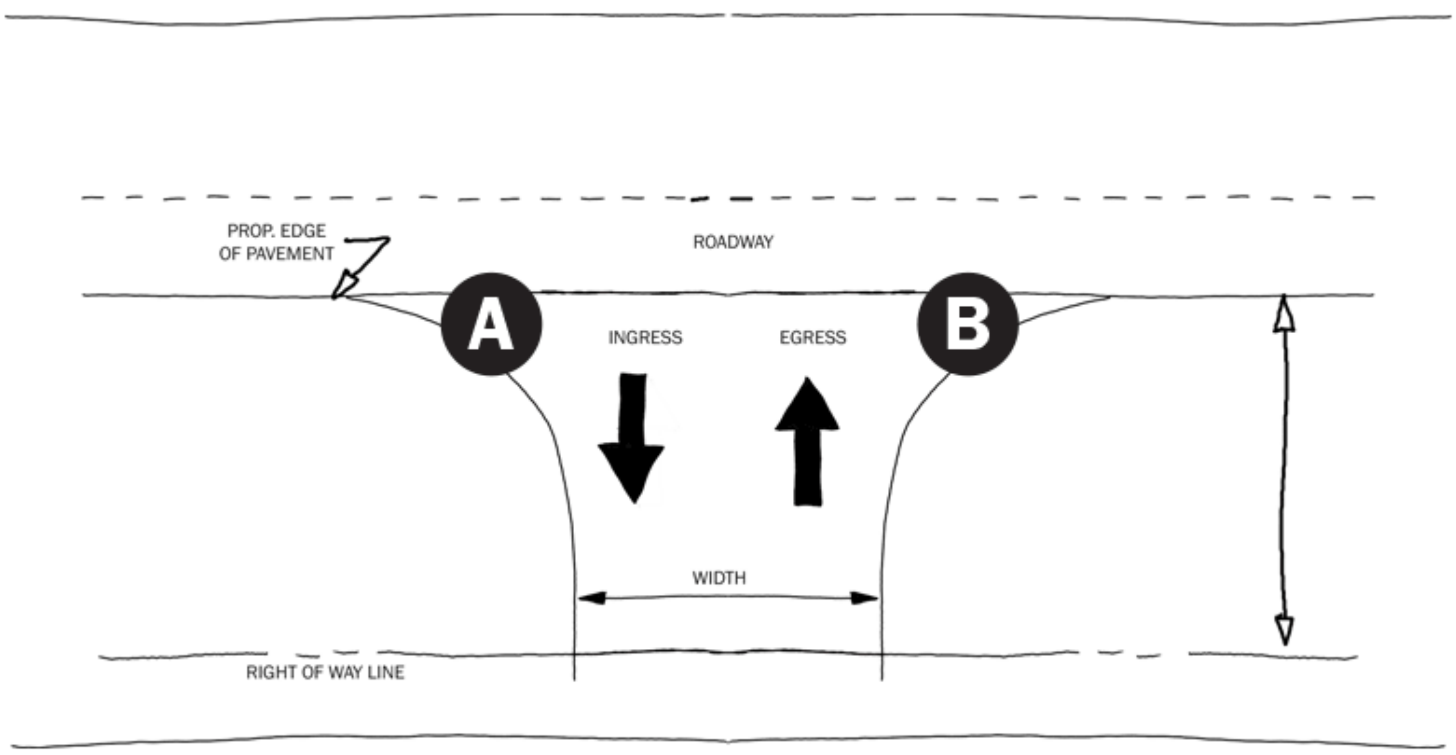
Kibana UI will be reachable at container port 5601 and corresponding host/ Node Port combination. When you begin, there won't be any data in Kibana (which is expected as you have not pushed any data).

TEMP_OTHERS

Ingress and Egress

What is the difference between ingress and egress?
• Ingress refers to the right to enter a property, while egress refers to the right to exit a property

• Ingress in the world of networking is traffic that enters the boundary of a network, while Egress implies traffic that exits an entity or a network boundary.



The definitions of Egress and Ingress for the cloud:
◇ Egress in the world of networking implies traffic that exits an entity or a network boundary,

◇ Ingress is traffic that enters the boundary of a network.

◇ While in service provider types of the network this is clear, in the case of datacenter or cloud it is slightly different.

◇ In the cloud, Egress still means traffic that is leaving from inside the private network out to the public internet, but Ingress means something slightly different.

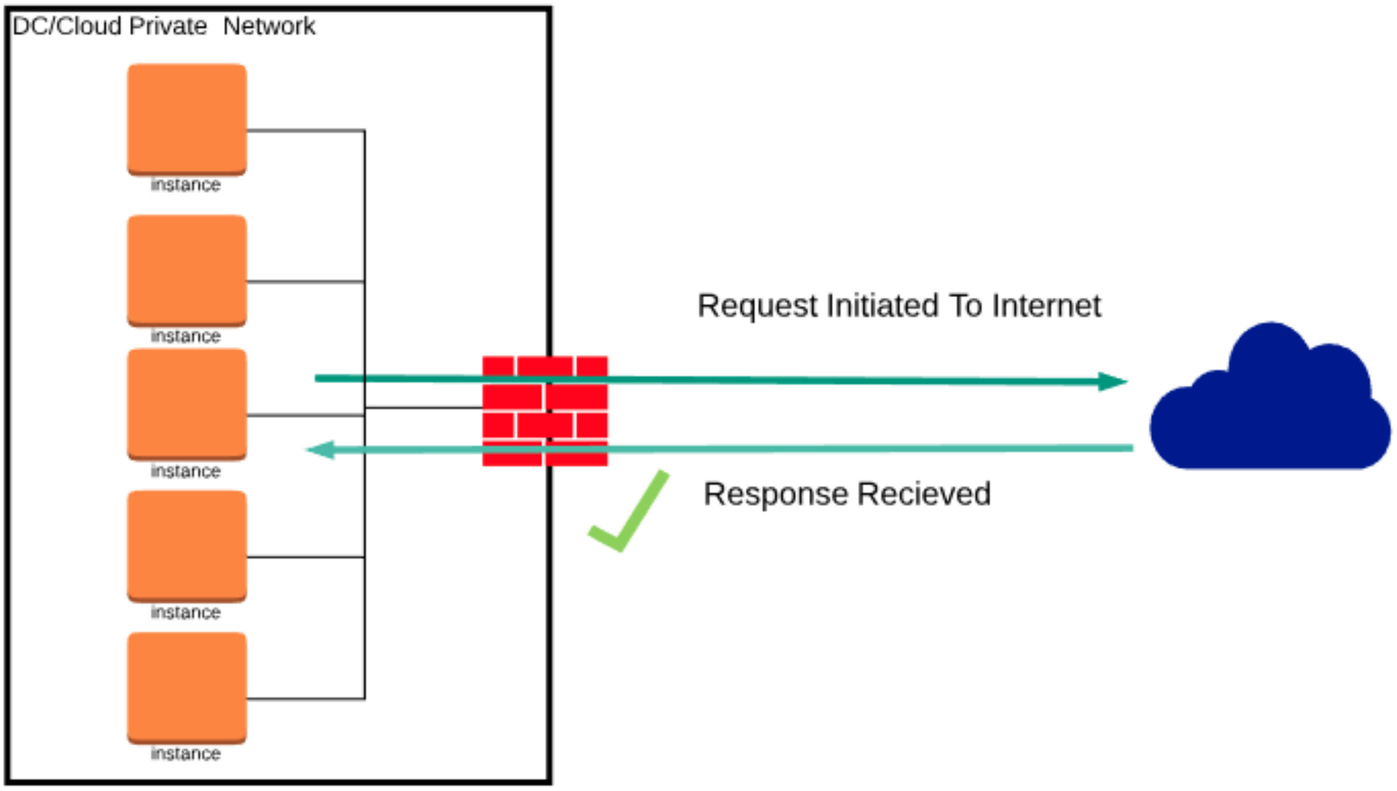
◇ To be clear, private networks here refers to resources inside the network boundary of a data center or cloud environment and its IP space is completely under the control of an entity who operates it.

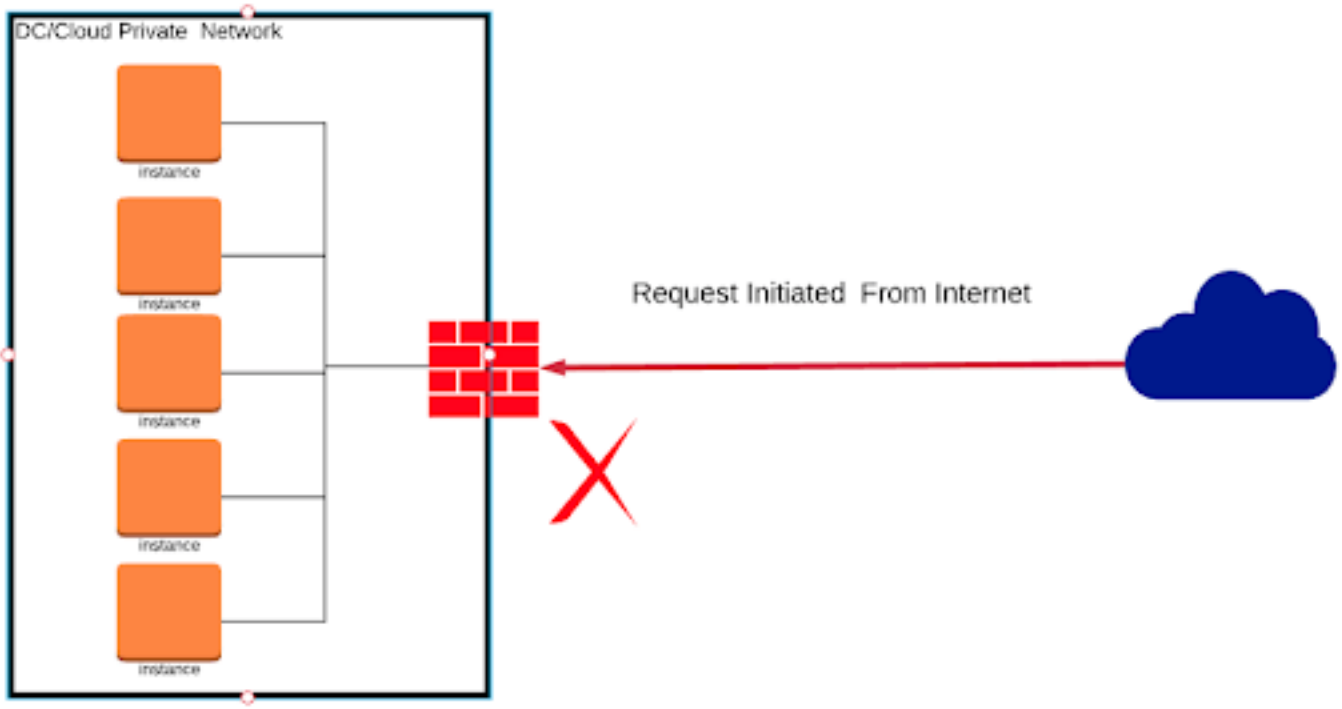
◇ Since traffic often is translated using NAT in and out of a private network like the cloud, a response back from a public endpoint to a request that was initiated inside the private network is not considered Ingress.

◇ If a request is made from the private network out to a public IP, the public server/ endpoint responds back to that request using a port number that was defined in the request, and firewall allows that connection since it's aware of an initiated session based on that port number. See picture below for reference.

Egress:
◇ Ingress refers to unsolicited traffic sent from an address on the public internet to the private network — it is not a response to a request initiated by an inside system.

◇ In this case, firewalls are designed to decline this request unless there is a specific policy and configuration that allows ingress connections. See picture below for reference.





Openshift container platform egress router pod

◇ The OpenShift Container Platform egress router pod redirects traffic to a specified remote server, using a private source IP address that is not used for any other purpose. This allows you to send network traffic to servers that are set up to allow access only from specific IP addresses

◇ The egress router pod is not intended for every outgoing connection. Creating large numbers of egress router pods can exceed the limits of your network hardware. For example, creating an egress router pod for every project or application could exceed the number of local MAC addresses that the network interface can handle before reverting to filtering MAC addresses in software.

Egress router modes

◇ In redirect mode, an egress router pod sets up iptables rules to redirect traffic from its own IP address to one or more destination IP addresses. Client pods that need to use the reserved source IP address must be modified to connect to the egress router rather than connecting directly to the destination IP.