CitiusTech

Kubernetes

Version 1.0

December 2020

Agenda

- **Overview**
- Kubernetes Architecture and Components
- Node/Minion Components
- Introduction to YAML
- Pods
- ReplicaSet and Services
- Deployments (Updates/Rollbacks)
- Kubernetes Networking
- Resource Requirements

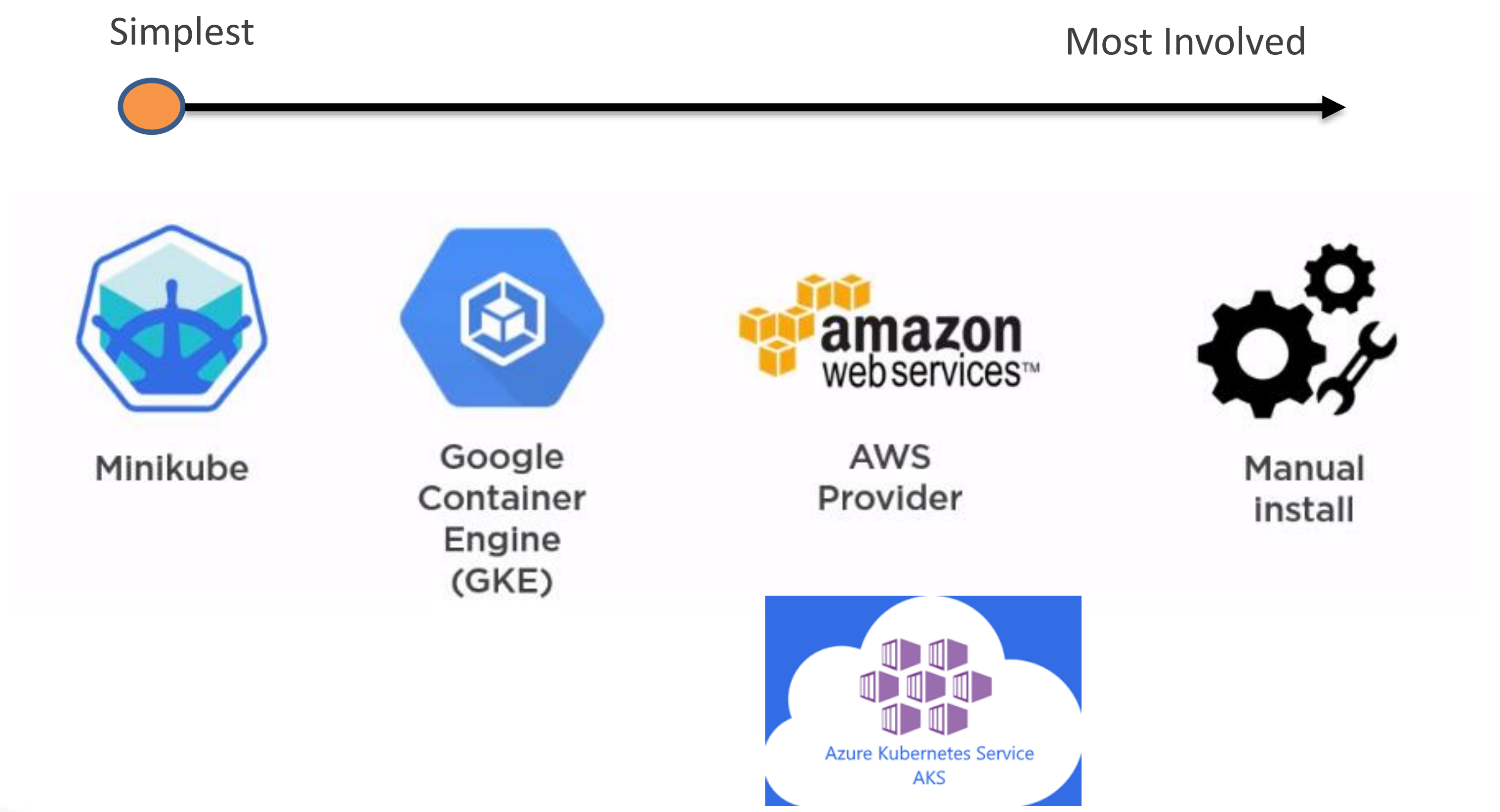
Overview (1/2)

- Modern application demands
  - 24X7 Availability
  - Self-healing capability
  - Horizontal scaling
    - Services need to be easily scaled up and down automatically based upon CPU utilization, or manually using a command.
  - Service discovery and load balancing
  - Application upgrades and rollbacks
    - Applications can be upgraded to a newer version without an impact to the existing one. If something goes wrong, rollback can be done.

Overview (2/2)

- Kubernetes is a **portable, extensible**, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.
- **Container Orchestration Tool:** As a Kubernetes orchestration it allows you to build application services that span multiple containers, schedule containers across a cluster, scale those containers, and manage their health over time

Kubernetes Implementations



Why you need Kubernetes?

- 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.
- 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.

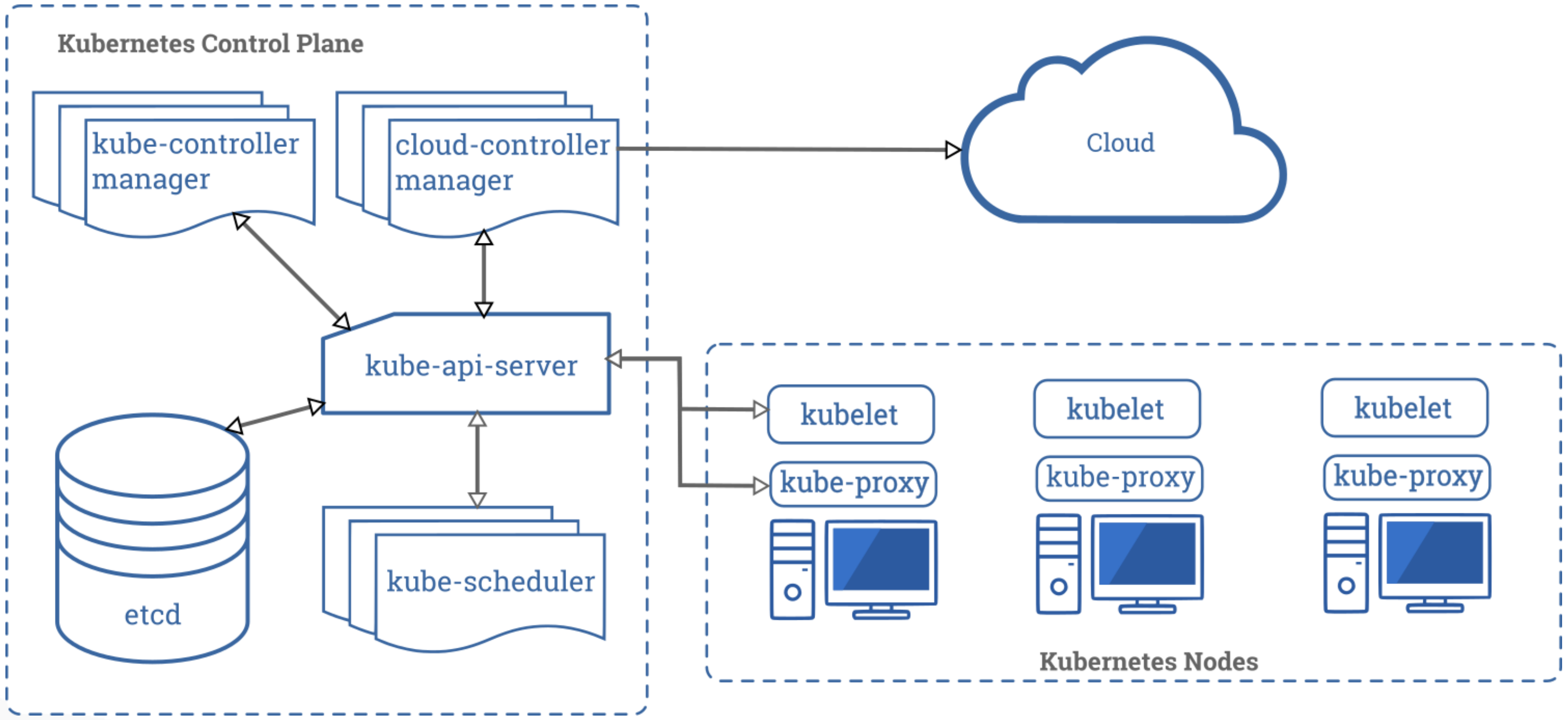
Kubernetes provides following:					
Service discovery and load balancing	**Storage Orchestration**	**Automated rollouts and rollbacks**	**Automatic bin packaging**	**Self-healing**	**Secret and configuration management**

Agenda

- Overview
- **Kubernetes Architecture and Components**
- Node/Minion Components
- Introduction to YAML
- Pods
- ReplicaSet and Services
- Deployments (Updates/Rollbacks)
- Kubernetes Networking
- Resource Requirements

Kubernetes Architecture and Components (1/5)

- A Kubernetes cluster consists of the components that represent the control plane and a set of machines called nodes.



https://d33wubrfki0l68.cloudfront.net/7016517375d10c702489167e704dcb99e570df85/7bb53/images/docs/components-of-kubernetes.png

Kubernetes Architecture and Components (2/5)

- 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 that are the components of the application workload.
- 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.

Kubernetes Architecture and Components (3/5)

Control Plane Components

- The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (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

kube-apiserver

- The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.
  - Front end to the control plane
  - Exposes the API (REST)
  - Consumes JSON (via manifest file)

Kubernetes Architecture and Components (4/5)

Cluster Store (etcd)

- Consistent and highly-available key value store used as Kubernetes backing store for all cluster data.
  - Persistence storage
  - Cluster state and Config
  - Uses etcd
  - Distributed, consistent, watchable
  - The “Source of truth”

kube-scheduler

- Control plane component that watches for newly created Pods with no assigned node and selects a node for them to run on.
  - Watches apiserver for new pods
  - Assigns work to nodes
    - Affinity/anti-affinity
    - Constraints
    - Resources
    - Data locality, inter-workload interference

Kubernetes Architecture and Components (5/5)

kube-controller-manager (Watches for changes and helps to maintain desired state)

- 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.
- These controllers are :
  - **Node controller:** Responsible for noticing and responding when nodes go down
  - **Node controller:** Responsible for maintaining the correct number of pods for every replication controller object in the system
  - **Endpoints controller:** Responsible for maintaining the correct number of pods for every replication controller object in the system
  - **Service Account & Token controllers:** Create default accounts and API access tokens for new namespaces

Summary

- Master- Control Plane Components

API Server	Cluster Store	Scheduler	Controller Manager
Central	Persists State	Watches API Server	Controller Loops
Simple	Key-Value	Schedules Pods	Lifecycle functions and desired state
RESTful	etcd	Resources	Watch and Update the API Server
Updates etcd	Watch	Respects constraints	ReplicaSet

Agenda

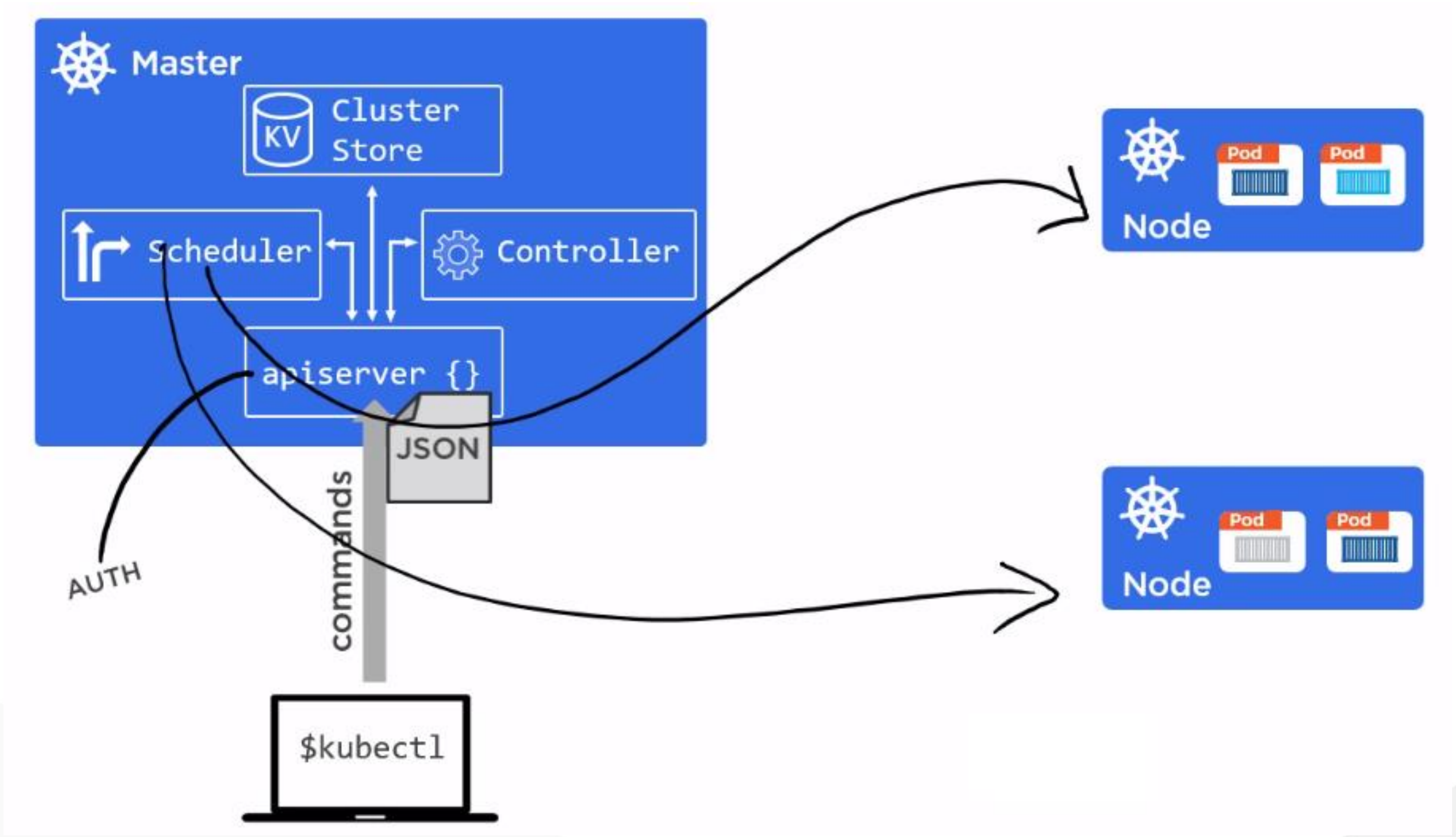
- Overview
- Kubernetes Architecture and Components
- **Node/Minion Components**
- Introduction to YAML
- Pods
- ReplicaSet and Services
- Deployments (Updates/Rollbacks)
- Kubernetes Networking
- Resource Requirements

Node/Minion Components (1/4)

Nodes a.k.a “Minions”

The Kubernetes Workers

Node/Minion Components (2/4)



Node/Minion Components (3/4)

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

Kubelet:

- An agent that runs on each node in the cluster. It ensures that containers are running in a Pod.
- 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 main Kubernetes Agent
    - Registers Node with Cluster
    - Watches apiserver
    - Instantiates pods
    - Reports back to master
    - Exposes endpoints on :10255

Node/Minion Components (4/4)

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.
- Key responsibilities are :
    - Kubernets Networking
    - Pod IP Addresses
        - All containers in a pod share a single IP
    - Load balances across all pods in a service

Container Engine

- Does Container Management:
    - Pulling images
    - Starting and stopping containers
- Pluggable:
    - Usually Docker
    - Can be rkt

Summary

- Node/Minion Components

Kubelet	Kube-proxy	Container Runtime
Monitor API Server for Changes	Network proxy iptables	Downloads images & runs containers
Responsible for Pod Lifecycle	Implements Service	Container Runtime Interface
Reports Node & Pod State	Routing traffic to Pods	Docker
Pod liveness probes	Load balancing	Many Others ..

Agenda

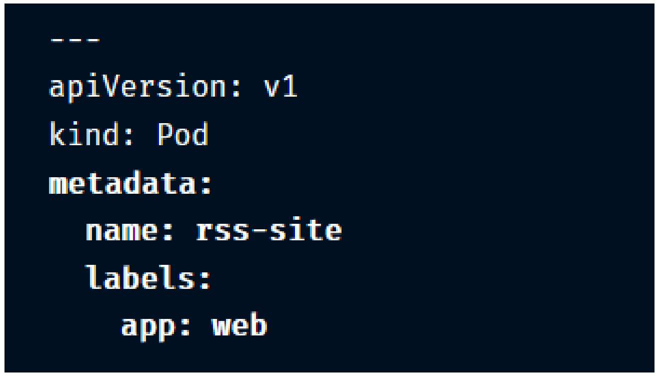
- Overview
- Kubernetes Architecture and Components
- Node/Minion Components
- **Introduction to YAML**
- Pods
- ReplicaSet and Services
- Deployments (Updates/Rollbacks)
- Kubernetes Networking
- Resource Requirements

Introduction to YAML (1/4)

- YAML, which stands for Yet Another Markup Language, or YAML Ain't Markup Language (depending who you ask) is a human-readable text-based format for specifying configuration-type information.
- Using YAML for K8s definitions gives a number of advantages, including:
  - **Convenience:** You'll no longer have to add all of your parameters to the command line
  - **Maintenance:** YAML files can be added to source control, so you can track changes
  - **Flexibility:** You'll be able to create much more complex structures using YAML than you can on the command line

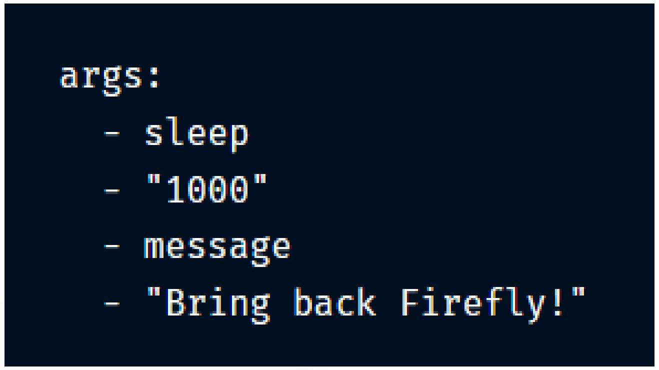
Introduction to YAML (2/4)

- **YAML Maps:** Key can have value and another Map as well



```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: rss-site
  labels:
    app: web
```

- **YAML lists** are literally a sequence of objects :



```yaml
args:
  - sleep
  - "1000"
  - message
  - "Bring back Firefly!"
```

- As you can see here, you can have virtually any number of items in a list, which is defined as items that start with a dash (-) indented from the parent.

Introduction to YAML (3/4)

- And members of the list can also be maps

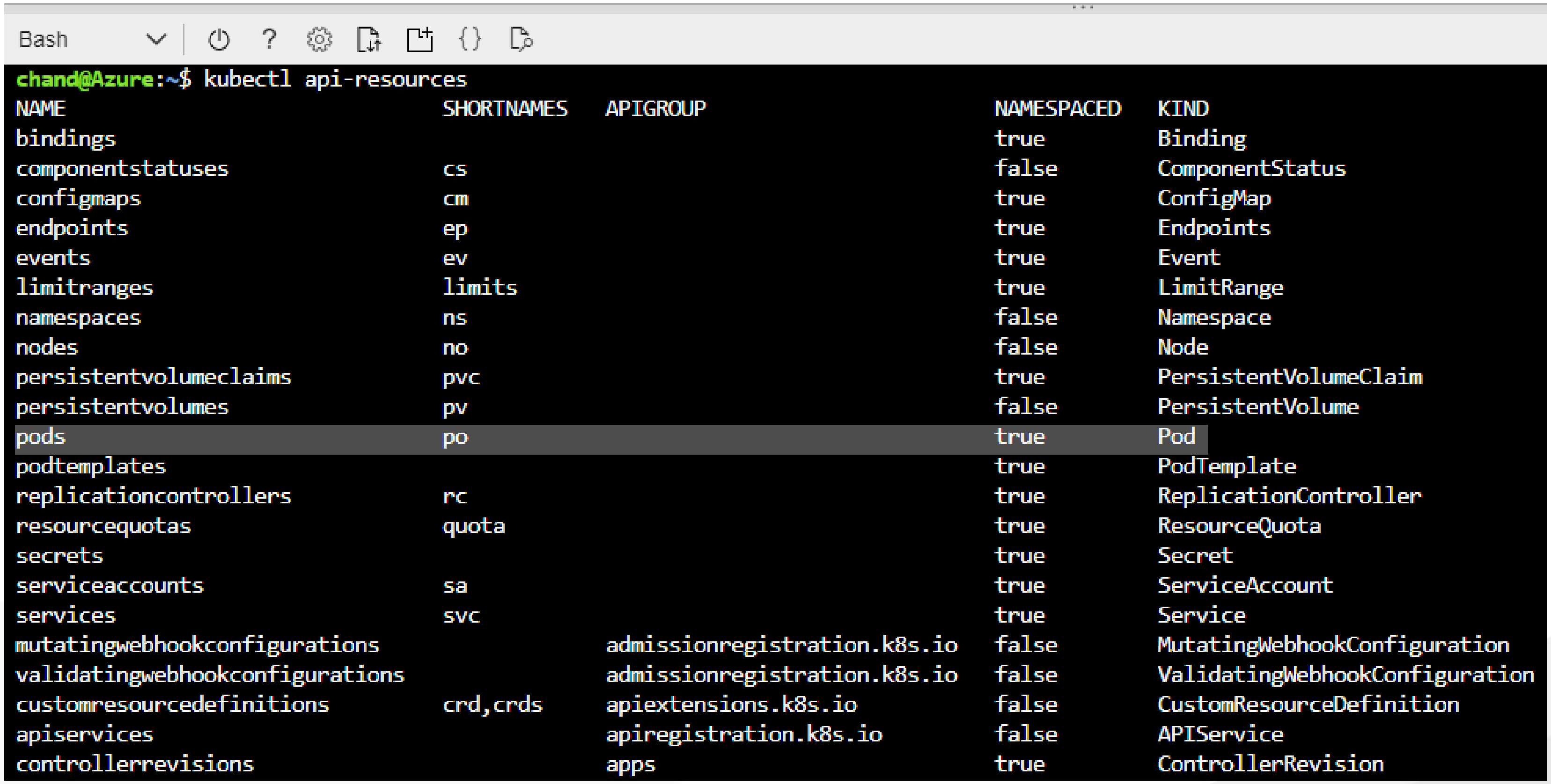
```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: rss-site
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: rss-reader
      image: nickchase/rss-php-nginx:v1
      ports:
        - containerPort: 88
```

Introduction to YAML (4/4)

- So to be précised we have:
  - maps, which are groups of name-value pairs
  - lists, which are individual items
  - maps of maps
  - maps of lists
  - lists of lists
  - lists of maps
- Whatever structure you want to put together, you can do it with those two structures (List and Maps)

Kubernetes Primitive Objects

- Kubernetes API Primitives are also called Kubernetes Objects . These are data Objects that represent the state of the Cluster. E.g. Pod, Node, Service etc.
- The **kubectl api-resources** command will list the object types currently available to the cluster



```
chand@Azure:~$ kubectl api-resources
NAME                             SHORTNAMES   APIGROUP                       NAMESPACED   KIND
bindings                                                                     true         Binding
componentstatuses                cs                                          false        ComponentStatus
configmaps                       cm                                          true         ConfigMap
endpoints                        ep                                          true         Endpoints
events                           ev                                          true         Event
limitranges                      limits                                      true         LimitRange
namespaces                       ns                                          false        Namespace
nodes                            no                                          false        Node
persistentvolumeclaims           pvc                                         true         PersistentVolumeClaim
persistentvolumes                pv                                          false        PersistentVolume
pods                             po                                          true         Pod
podtemplates                                                                 true         PodTemplate
replicationcontrollers           rc                                          true         ReplicationController
resourcequotas                   quota                                       true         ResourceQuota
secrets                                                                      true         Secret
serviceaccounts                  sa                                          true         ServiceAccount
services                         svc                                         true         Service
mutatingwebhookconfigurations                 admissionregistration.k8s.io   false        MutatingWebhookConfiguration
validatingwebhookconfigurations               admissionregistration.k8s.io   false        ValidatingWebhookConfiguration
customresourcedefinitions        crd,crds     apiextensions.k8s.io           false        CustomResourceDefinition
apiservices                                   apiregistration.k8s.io         false        APIService
controllerrevisions                           apps                           true         ControllerRevision
```

Agenda

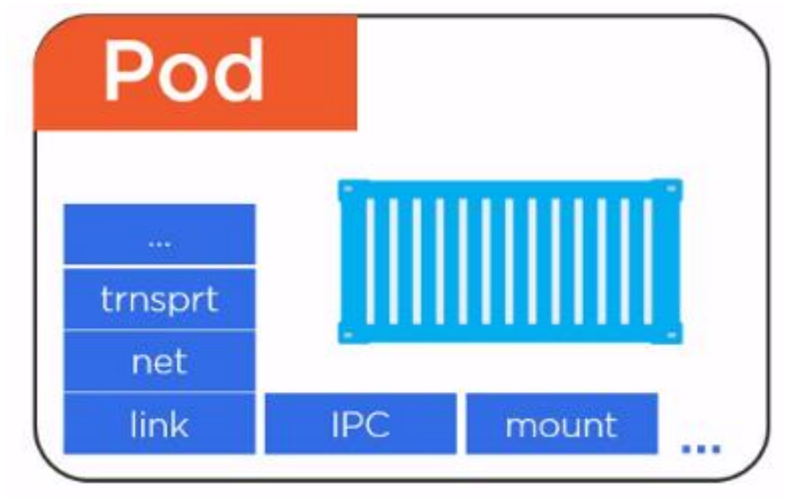
- Overview
- Kubernetes Architecture and Components
- Node/Minion Components
- Introduction to YAML
- **Pods**
- ReplicaSet and Services
- Deployments (Updates/Rollbacks)
- Kubernetes Networking
- Resource Requirements

Pods (1/2)

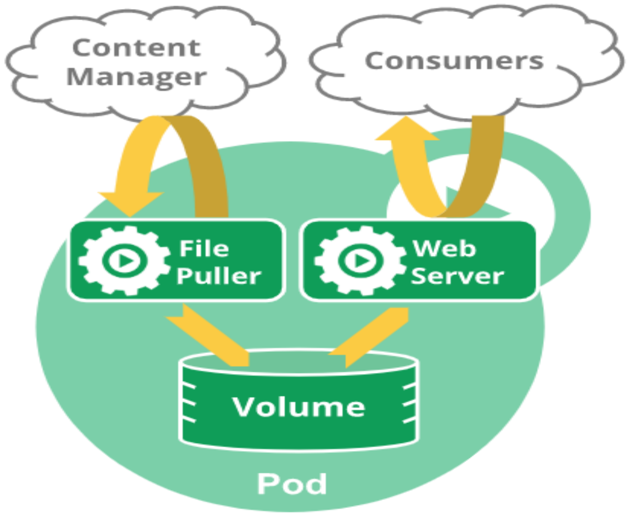
Pods are the smallest deployable units of computing that can be created and managed in Kubernetes.

- It is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers.
- While Kubernetes supports more container runtimes than just Docker, Docker is the most known runtime, and it helps to describe Pods in Docker terms.
- The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container.
- All Containers within a Pod share an IP address and port space, and can find each other via localhost
- Containers in different Pods have distinct IP addresses and can not communicate by IPC without special configuration. These containers usually communicate with each other via Pod IP addresses.

Pods (2/2)



Single Pod



A multi-container Pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers.

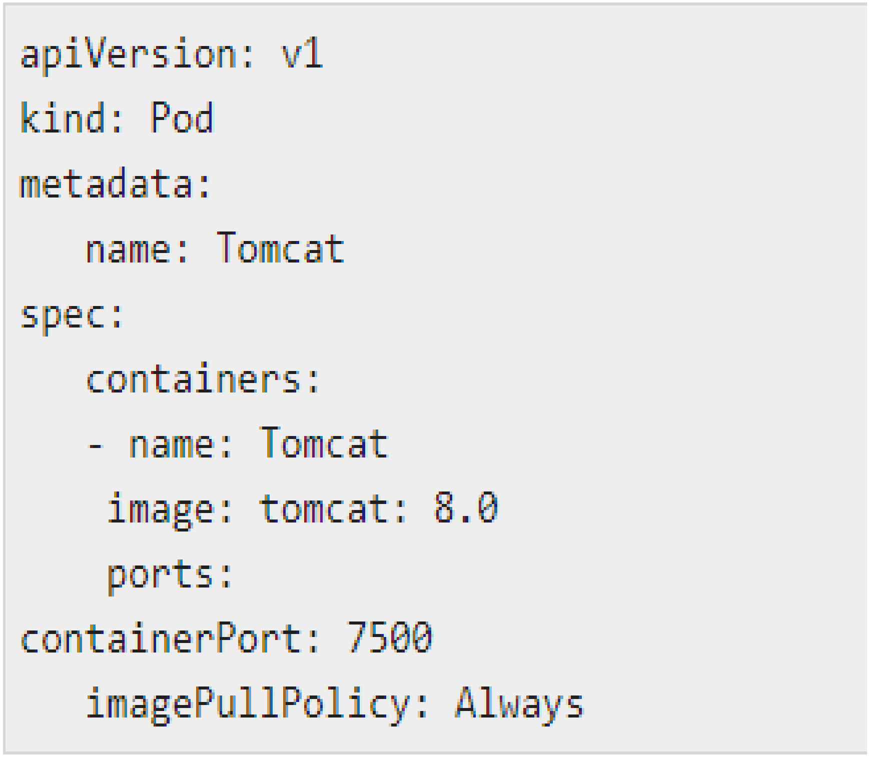
https://kubernetes.io/docs/concepts/workloads/pods/pod/

Pod LifeCycle

The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle. Here are the possible values for phase:

Value	Description
Pending	The Pod has been accepted by the Kubernetes system, but one or more of the Container images has not been created. This includes time before being scheduled as well as time spent downloading images over the network, which could take a while.
Running	The Pod has been bound to a node, and all of the Containers have been created. At least one Container is still running or is in the process of starting or restarting.
Succeeded	All Containers in the Pod have terminated in success and will not be restarted.
Failed	All Containers in the Pod have terminated, and at least one Container has terminated in failure. That is, the Container either exited with non-zero status or was terminated by the system.
Unknown	For some reason the state of the Pod could not be obtained, typically due to an error in communicating with the host of the Pod.

Pod : Example



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

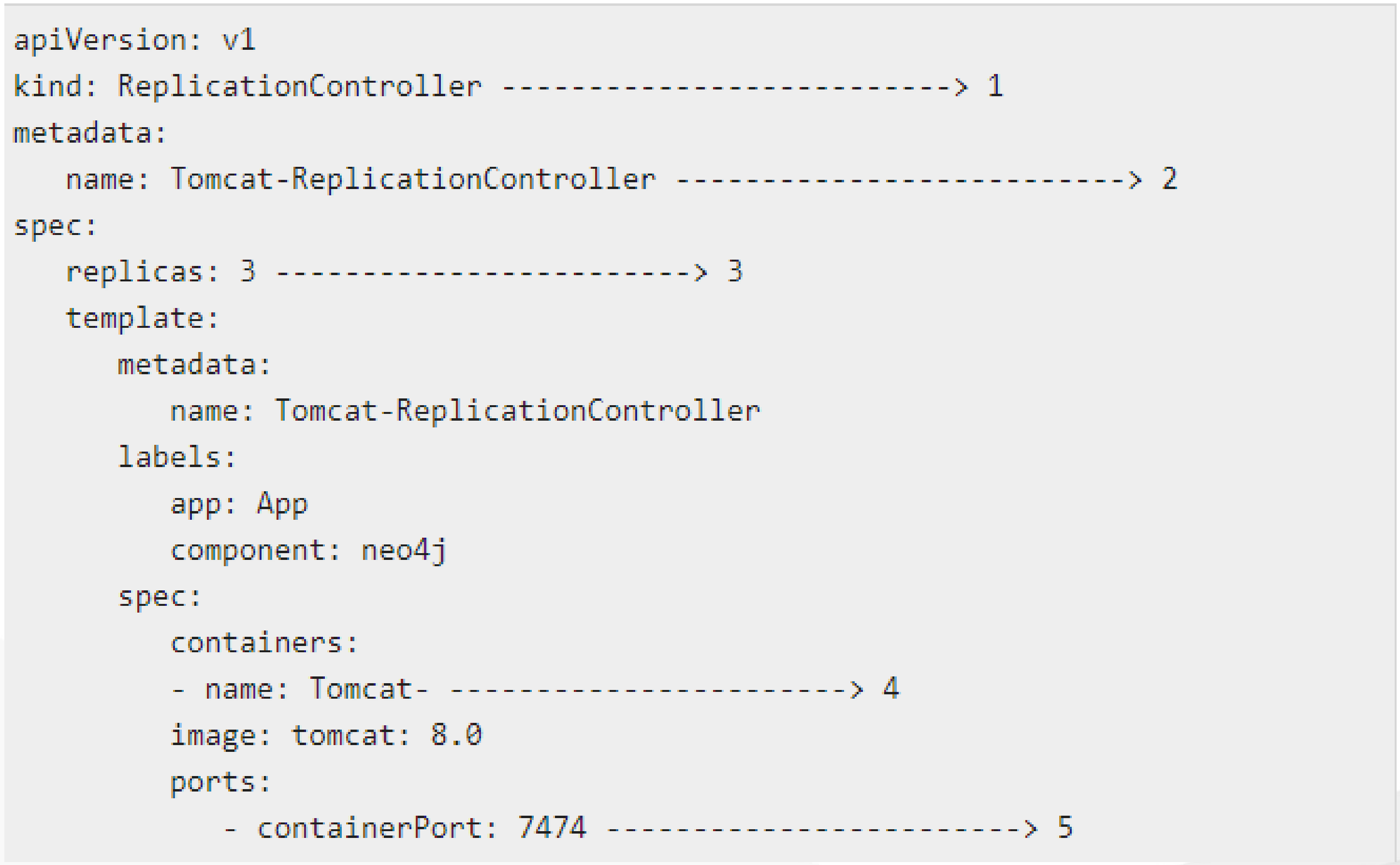
Examples	Description
apiVersion	Which version of the Kubernetes API you're using to create this object
kind	What kind of object you want to create
metadata -	Data that helps uniquely identify the object, including a name string, UID, and optional namespace
spec	What state you desire for the object

Agenda

- Overview
- Kubernetes Architecture and Components
- Node/Minion Components
- Introduction to YAML
- Pods
- **ReplicaSet and Services**
- Deployments (Updates/Rollbacks)
- Kubernetes Networking
- Resource Requirements

ReplicaSet (1/2)

- A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.



```
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
```

CitiusTech

ReplicaSet (2/2)

Examples	Description
Kind: ReplicationController	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 ensuring 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

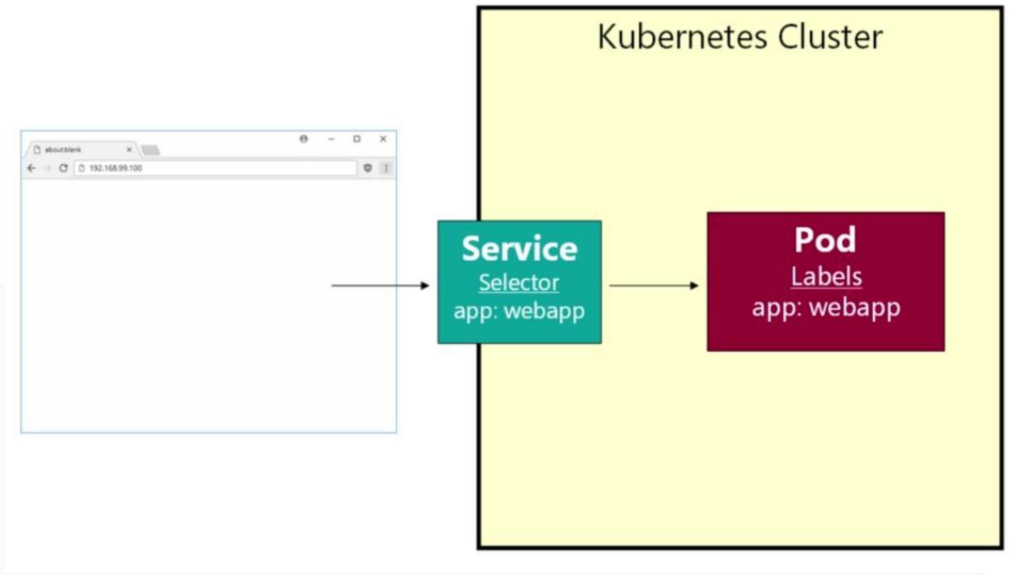
Services (1/2)

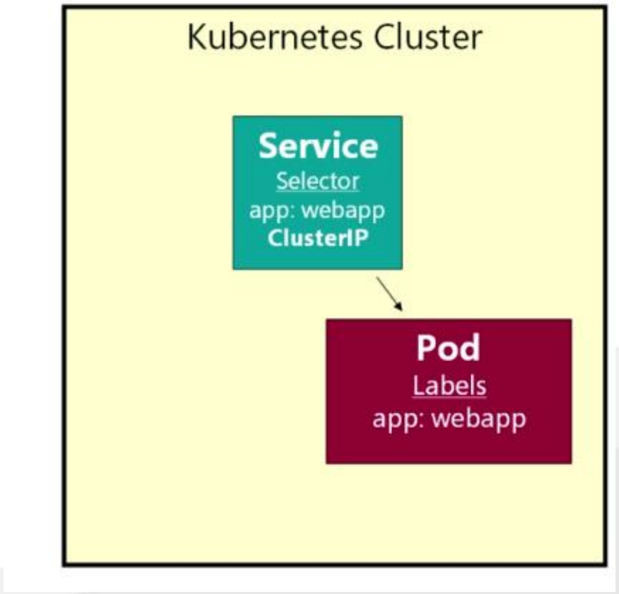
- Motivation : How do we access our app?
  - From outside the cluster ?
  - From inside the cluster ?
- Kubernetes Pods are mortal. They are born and when they die, they are not resurrected. If you use a deployment to run your app, it can create and destroy Pods dynamically.
- Each Pod gets its own IP address, however in a deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.
- This leads to a problem: if some set of Pods (call them “backends”) provides functionality to other Pods (call them “frontends”) inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?
- **Service:** An abstract way to expose an application running on a set of Pods as a network service.
- 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.

Services (2/2)

Types of Services

- **ClusterIP :** Stable Internal cluster IP. This helps in restricting the service within the cluster. It exposes the service within the defined Kubernetes cluster.
- **NodePort :** Exposes the app outside of the cluster by adding a cluster-wide port on top of ClusterIP. The service can be accessed from outside the cluster using the **NodeIP:nodePort**.
- **LoadBalancer:** Integrates NodePort with cloud-based load balancers. **NodePort** and **ClusterIP** services are created automatically to which the external load balancer will route.





Agenda

- Overview
- Kubernetes Architecture and Components
- Node/Minion Components
- Introduction to YAML
- Pods
- ReplicaSet and Services
- **Deployments (Updates/Rollbacks)**
- Kubernetes Networking
- Resource Requirements

Deployments (Updates/Rollbacks) (1/5)

- A Deployment provides declarative updates for Pods and ReplicaSets.
- You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.
- They manage the deployment of replica sets. They have the capability to update the replica set and are also capable of rolling back to the previous version.
- You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.
- They provide many updated features of **matchLabels** and **selectors.**
- We have got a new controller in the Kubernetes master called the deployment controller which has the capability to change the deployment midway.

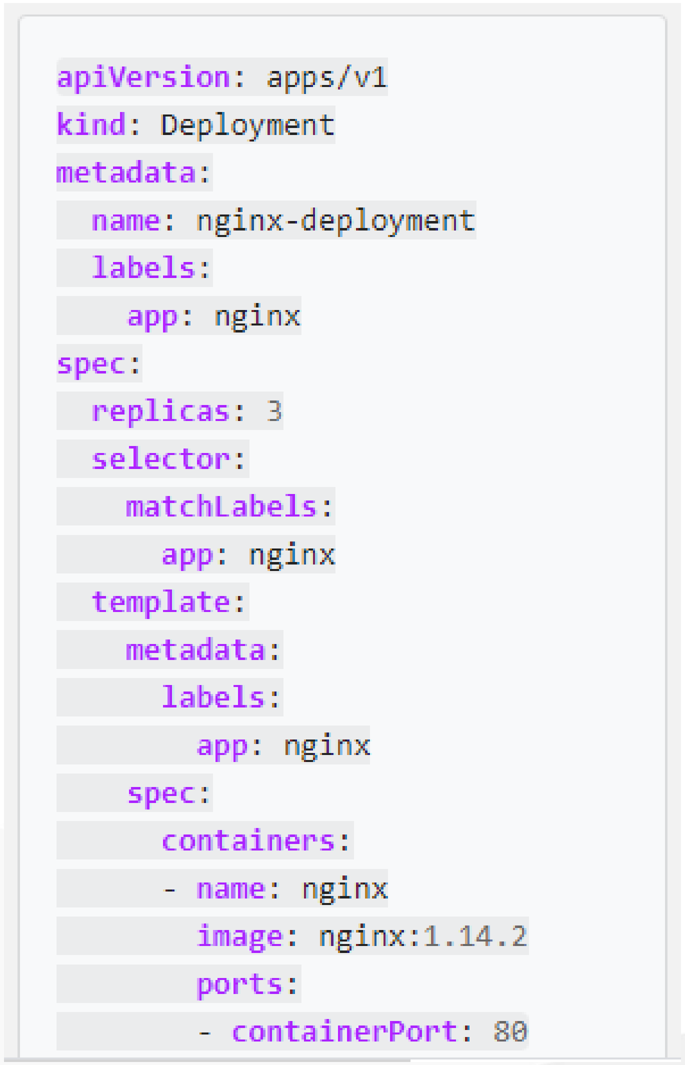
Deployments (Updates/Rollbacks) (2/5)

The following are typical use cases for Deployments:

- Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or fails.
- Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
- Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- Scale up the Deployment to facilitate more load
- Pause the Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- Use the status of the Deployment as an indicator that a rollout has stuck.
- Clean up older ReplicaSets that you don't need anymore.

Deployments (Updates/Rollbacks) (3/5)

The following is an example of a Deployment. It creates a ReplicaSet to bring up three nginx Pods:



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Deployments (Updates/Rollbacks) (4/5)

Deployment strategies help in defining how the new RS(ReplicaSet) should replace the existing RC.

- **Recreate** − This feature will kill all the existing RS 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 RS 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.

Deployments (Updates/Rollbacks) (5/5)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: Tomcat-ReplicaSet
spec:
  replicas: 3
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
      metadata:
          lables:
             app: Tomcat-ReplicaSet
             tier: Backend
  spec:
     containers:
         - name: Tomcat
           image: tomcat: 8.0
           ports:
               - containerPort: 7474
```

Agenda

- Overview
- Kubernetes Architecture and Components
- Node/Minion Components
- Introduction to YAML
- Pods
- ReplicaSet and Services
- Deployments (Updates/Rollbacks)
- **Kubernetes Networking**
- Resource Requirements

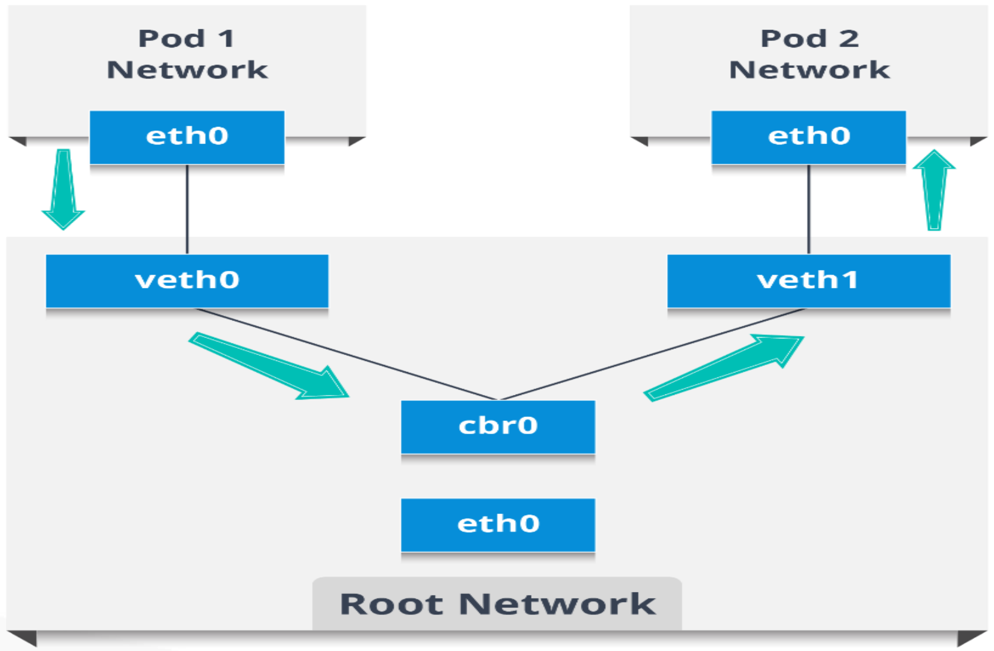
Kubernetes Networking (1/3)

- Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:
    - Highly-coupled container-to-container communications: this is solved by pods and localhost communications.
    - Pod-to-Pod communications : this is taken care by host machine
    - Pod-to-Service communications: this is covered by services.
    - External-to-Service communications: this is covered by services.
- The communication between pods, services and external services to the ones in a cluster brings in the concept of Kubernetes networking.
- In Kubernetes, every pod has its own routable IP address. Kubernetes networking – through the network plug-in that is required to install (e.g. Calico, Flannel, Weave…) takes care of routing all requests internally between hosts to the appropriate pod.
- External access is provided through a service, load balancer, or ingress controller, which Kubernetes routes to the appropriate pod.

Kubernetes Networking (2/3)

How Pods communicate ? There are 2 types of communication. The **inter-node communication** and the **intra-node communication**.

Intra-node Pod Network



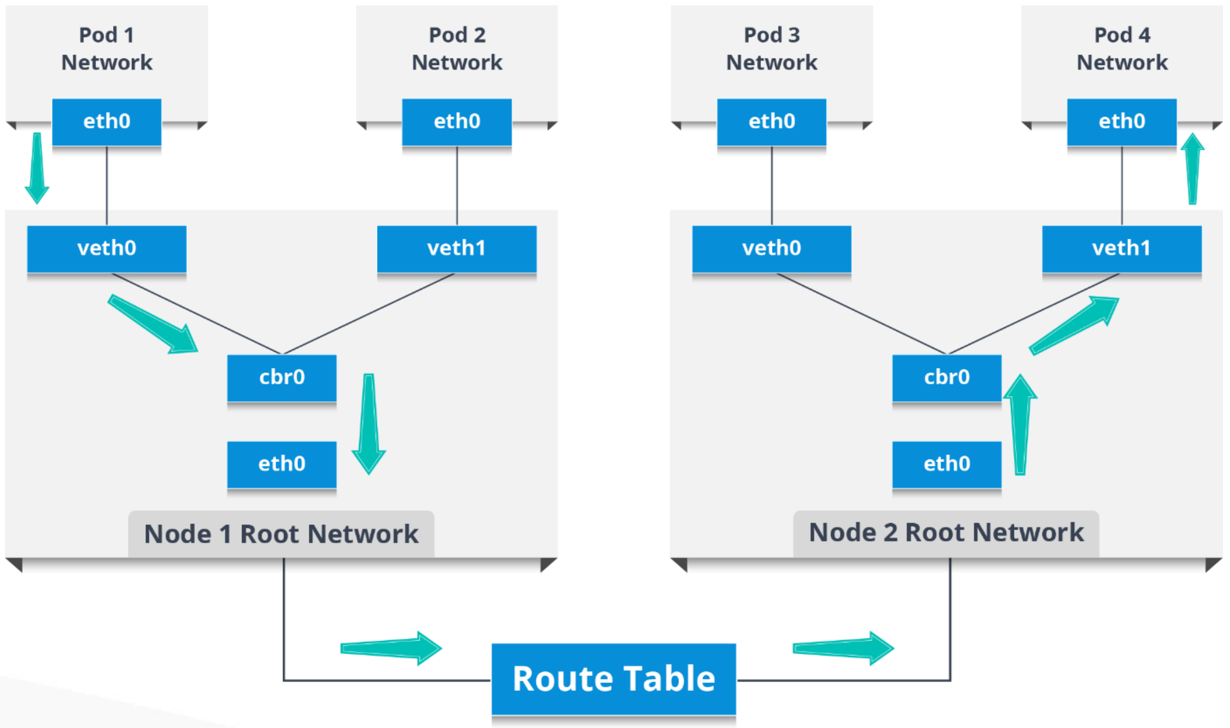
- Above Diagram shows how pods on the same node are able to talk to each other
- Each pod has its own netns(network namespace), with a virtual ethernet pair connecting it to the root netns. This is basically a pipe-pair with one end in root netns, and other in the pod netns.

https://www.edureka.co/blog/kubernetes-networking/

Kubernetes Networking (3/3)

Inter-node communication

- Consider two nodes having various network namespaces, network interfaces, and a Linux bridge.



CitiusTech

Kubernetes Network Model

- Every Pod gets its own IP address. This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports.
- This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes Namespaces

- Namespaces provide a way to keep our objects organized within the cluster. Every object belongs to a namespace. When no namespace is specified, the cluster will assume the default namespace.
- When creating an object , you can assign it to a namespace by specifying a namespace in the metadata :

```
apiVersion: v1
kind: Pod
metadata:
  name: my-ns-pod
  namespace: my-ns
  labels:
    app: myappspec:
spec:
containers:
   name: myapp-container
   image: busybox
   command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600’]
```

- Use the -n flag to specify a namespace when using commands like kubectl get Kubectl get pods –n my-ns

ConfigMaps

- Management of configuration data is one of the challenges involved in building and maintaining complex application infrastructures. Luckily, Kubernetes offers functionality that helps to maintain application configurations in the form of ConfigMaps.
- A ConfigMaps is a Kubernetes Object that stores configuration data in a key-value format. This Configuration data can then be used to configure software running in a container by referencing the ConfigMap in the Pod spec.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config-map
data:
  myKey: myValue
  anotherKey: anotherValue
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-configmap-pod
spec:
  containers:
  .
  .
    env:
    - name: MY_VAR
      valueFrom:
        configMapKeyRef:
          name: my-config-map
          key: myKey
```

Secrets

- Secrets are pieces of sensitive Information stored in Kubernetes cluster such as passwords, tokens and keys.
- If container needs a sensitive piece of information such as password it is more secure to store it as a secret than storing it in a pod or in the container itself.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
stringData:
  myKey: myPassword
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-secret-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo Hello, Kubernetes! 
&& sleep 3600"]
    env:
    - name: MY_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: myKey
```

SecurityContexts

- Occasionally, it's necessary to customize how containers interact with the underlying security mechanisms present on the operating systems of Kubernetes nodes.
- A Pods Security Context defines privilege and access control settings for a pod. If a container needs a special operating settings for a pod or special operating system level permissions, we can provide them using securityContext.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-securitycontext-pod
spec:
  securityContext:
    runAsUser: 2001
    fsGroup: 3001
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "cat /message/message.txt && sleep 3600"]
```

```
sudo useradd -u 2001 container-user-1
sudo groupadd -g 3001 container-group-1
```

Note*: Above spec will cause the container to run as the OS user with and ID of 2001, and the containers will run as the group with an ID of 3001

Agenda

Resource Requirements (1/2)

- Kubernetes is a powerful tool for managing and utilizing available resources to run containers. Resource requests and limits provide a great deal of control over how resources will be allocated.
- Kubernetes allow us to specify the resource requirements of a container in the pod spec. A container's memory and CPU requirements are defined in terms of **resource requests** and **limits.**
- **Resource Request :** The amount of resources necessary to run a container
    - A pod will only be a run on node that has enough available resources to run the pod’s containers.
- **Resource Limit :** A maximum value for the resource usage of a container

Resource Requirements (2/2)

- Sample yaml file defining Resource Requirements

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-resource-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m“
```

Memory is measured in bytes. 64Mi means 64 Megabytes. CPU is measured in “Cores”. 250 m means 250 milliCPU’s, or 0.25 CPU cores

Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email ct-univerct@citiustech.com

www.citiustech.com

CitiusTech