Desenvolvimento de Aplicações com Arquitetura Baseada em Microservices

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: @vinicius3w :: assertlab.com

[IF1007] - Tópicos Avançados em SI 4
https://github.com/vinicius3w/if1007-Microservices

AssertLab
Advanced Software and Systems Engineering Research Technologies

Licença do material

Este Trabalho foi licenciado com uma Licença

Creative Commons - Atribuição-NãoComercial-CompartilhaIgual 3.0 Não Adaptada

Mais informações visite

http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt

AssertLab

Resources

- There is no textbook required. However, the following are some books that may be recommended:
  - Introduction to Kubernetes - LinuxFoundationX - LFS158x
  - Kubernetes Ingress: NodePort, Load Balancers, and Ingress Controllers
  - Deploying Java Applications with Docker and Kubernetes
  - Deploying Java Applications with Kubernetes and an API Gateway
  - Kubernetes Documentation
  -

kubernetes

Introduction to Kubernetes

AssertLab
Advanced Software and Systems Engineering Research Technologies

Overview

- Do you need guidelines on how to start transforming your organization with Kubernetes and cloud native patterns?
- Would you like to simplify software container orchestration and find a way to grow your use of Kubernetes without adding infrastructure complexity?
- In this lecture, we'll discuss some of Kubernetes' basic concepts and talk about the architecture of the system, the problems it solves, and the model that it uses to handle containerized deployments and scaling.
- This lecture offers an introduction to Kubernetes and includes technical instructions on how to deploy a stand-alone and multi-tier application

Topics

- The origin, architecture, primary components, and building blocks of Kubernetes
- How to set up and access a Kubernetes cluster using Minikube
- Ways to run applications on the deployed Kubernetes environment and access the deployed applications
- Usefulness of Kubernetes communities and how you can participate

AssertLab
Advanced Software and Systems Engineering Research Technologies

Prerequisites

- Basic knowledge of container technologies, like Docker or rkt is required

Introduction

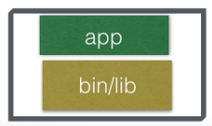
Warm up

- With container images, we confine the application code, its runtime, and all of its dependencies in a pre-defined format
- And, with container runtimes like runC, containerd, or rkt we can use those pre-packaged images, to create one or more containers
- All of these runtimes are good at running containers on a single host
- But, in practice, we would like to have a fault-tolerant and scalable solution, which can be achieved by creating a single controller/ management unit, after connecting multiple nodes together
- This controller/management unit is generally referred to as a Container Orchestrator.

AssertLab
Advanced Software and Systems Engineering Research Technologies

Warm up

- Containers are an application-centric way to deliver high-performing, scalable applications on the infrastructure of your choice
- With a container image, we bundle the application along with its runtime and dependencies
- We use that image to create an isolated executable environment, also known as container
- We can deploy containers from a given image on the platform of our choice, such as desktops, VMs, Cloud, etc.



Desktop | Dev VM | QA Env. | Public Cloud | Private Cloud | Customer Site

What Is Container Orchestration?

- In Development and Quality Assurance (QA) environments, we can get away with running containers on a single host to develop and test applications
- However, when we go to production, we do not have the same liberty, as we need to ensure that our applications:
  - Are fault-tolerant
  - Can scale, and do this on-demand
  - Use resources optimally
  - Can discover other applications automatically, and communicate with each other
  - Are accessible from the external world
  - Can update/rollback without any downtime.

What Is Container Orchestration?

- Container Orchestrators are the tools which group hosts together to form a cluster, and help us fulfill the requirements mentioned earlier

Container Orchestrators

- **Docker Swarm**: is a Container Orchestrator provided by Docker, Inc. It is part of Docker Engine
- **Kubernetes**: was started by Google, but now, it is a part of the Cloud Native Computing Foundation project
- **Mesos Marathon**: is one of the frameworks to run containers at scale on Apache Mesos
- **Amazon EC2 Container Service (ECS)**: is a hosted service provided by AWS to run Docker containers at scale on its infrastructrue
- **Hashicorp Nomad**: is the Container Orchestrator provided by HashiCorp

Why Use Container Orchestrators?

- Bring multiple hosts together and make them part of a cluster
- Schedule containers to run on different hosts
- Help containers running on one host reach out to containers running on other hosts in the cluster
- Bind containers and storage
- Bind containers of similar type to a higher-level construct, like services, so we don't have to deal with individual containers
- Keep resource usage in-check, and optimize it when necessary
- Allow secure access to applications running inside containers

Though we can argue that containers at scale can be maintained manually, or with the help of some scripts, Container Orchestrators can make things easy for operators. With all these built-in benefits, it makes sense to use Container Orchestrators to manage containers. In this lecture, we will explore Kubernetes.

Where to Deploy Container Orchestrators?

- Most Container Orchestrators can be deployed on the infrastructure of our choice
- We can deploy them on bare-metal, VMs, on-premise, or on a cloud of our choice
  - For example, Kubernetes can be deployed on our laptop/workstation, inside a company's datacenter, on AWS, on OpenStack, etc
  - There are even one-click installers available to setup Kubernetes on the Cloud, like Google Container Engine on Google Cloud, or Azure Container Service on Microsoft Azure
  - Similar solutions are available for other Container Orchestrators, as well.
- There are companies who o er managed Container Orchestration as a Service

Kubernetes

In this section, we will explain what Kubernetes is, its features, and the reasons why one should use it. We will explore the evolution of Kubernetes from Borg, which is a cluster manager created by Google.
We will also talk about the Cloud Native Computing Foundation (CNCF), which currently hosts the Kubernetes project, along with other cloud-native projects, like Prometheus, Fluentd, rkt, containerd, etc.

What Is Kubernetes?

- "Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications."
- **K8s** comes from the Greek word κυβερνήτης:, which means helmsman or ship pilot.
  - With this analogy in mind, we can think of **k8s** as the manager for shipping containers.

Kubernetes is also referred to as k8s, as there are 8 characters between k and s.

What Is Kubernetes?

- K8s is highly inspired by the Google Borg system, which we will explore in this lecture
  - It is an open source project written in the Go language, and licensed under the Apache License Version 2.0.
- k8s was started by Google and, with its v1.0 release in July 2015, Google donated it to the Cloud Native Computing Foundation (CNCF)
  - We will discuss more about CNCF a little later.
- Generally, k8s has new releases every three months.

From Borg to Kubernetes

- "Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines."
- For more than a decade, Borg was Google's secret to run containerized workloads in production
- Whatever services we use from Google, like Gmail, Drive, etc., they are all serviced using Borg.
- Some of the features/objects of k8s that can be traced back to Borg, or to lessons learnt from it, are:
  - API Servers
  - Pods
  - IP-per-Pod
  - Services
  - Labels

Some of the initial authors of Kubernetes were Google employees who have used Borg and developed it in the past. They poured in their valuable knowledge and experience while designing Kubernetes.

Kubernetes Features I

- **Automatic binpacking**: automatically schedules the containers based on resource usage and constraints, without sacrificing the availability
- **Self-healing**: automatically replaces and reschedules the containers from failed nodes. It also kills and restarts the containers which do not respond to health checks, based on existing rules/policy
- **Horizontal scaling**: can automatically scale applications based on resource usage like CPU and memory. In some cases, it also supports dynamic scaling based on customer metrics
- **Service discovery and Load balancing**: groups sets of containers and refers to them via a DNS name. This DNS name is also called a k8s **service**. K8s can discover these services automatically, and load-balance requests between containers of a given service.

Kubernetes Features II

- **Automated rollouts and rollbacks**: can roll out and roll back new versions/ configurations of an application, without introducing any downtime
- **Secrets and configuration management**: can manage secrets and configuration details for an application without re-building the respective images. With secrets, we can share confidential information to our application without exposing it to the stack configuration, like on GitHub
- **Storage orchestration**: with k8s and its plugins, we can automatically mount local, external, and storage solutions to the containers in a seamless manner, based on Software De ned Storage (SDS)
- **Batch execution**: besides long running jobs, k8s also supports batch execution

There are many other features besides the ones we just mentioned, and they are currently in alpha/beta phase. They will add great value to any k8s deployment once they become GA (generally available) features. For example, support for RBAC (Role-based access control).

Why Use Kubernetes?

- K8s is very portable and extensible and can be deployed on the environment of our choice, be it VMs, bare-metal, or public/private/hybrid/multi-cloud setups
- Also has a very modular and pluggable architecture
- We can write custom APIs or plugins to extend its functionalities
- K8s has a very thriving community across the world
  - It has more than 1350 contributors, who, over time, have done over 47,000 commits
  - There are meet-up groups in different cities which meet regularly to discuss about k8s and its ecosystem
  - There are Special Interest Groups (SIGs), which focus on special interests, such as scaling, bare-metal, networking, etc

Kubernetes Users

- With just a few years since its debut, many companies are running workloads using k8s
  - Box
  - eBay
  - Pearson
  - SAP
  - Wikimedia
  - IF1007
  - And many more

CLOUD NATIVE COMPUTING FOUNDATION

Cloud Native Computing Foundation (CNCF)

- The Cloud Native Computing Foundation (CNCF) is one of the projects hosted by The Linux Foundation
- CNCF aims to accelerate the adoption of containers, microservices, and cloud-native applications
- CNCF provides resources to each of the projects, but, at the same time, each project continues to operate independently under its pre-existing governance structure and with its existing maintainers

AssertLab Advanced Software and Systems Engineering Research Technologies

Currently, the following projects are part of CNCF:

- containerd for Container Runtime
- rkt for Container Runtime
- Kubernetes for Container Orchestration
- Linkerd for Service Mesh
- gRPC for Remote Procedure Call
- Container Network Interface (CNI) for Container Networking
- CoreDNS for Service Discovery
- Prometheus for Monitoring
- OpenTracing for Tracing
- Fluentd for Logging

As we can see, the current set of CNCF projects can cover the entire lifecycle of an application, from its execution using container runtimes, to its monitoring and logging. This is very important to meet the CNCF goal.

CNCF and Kubernetes

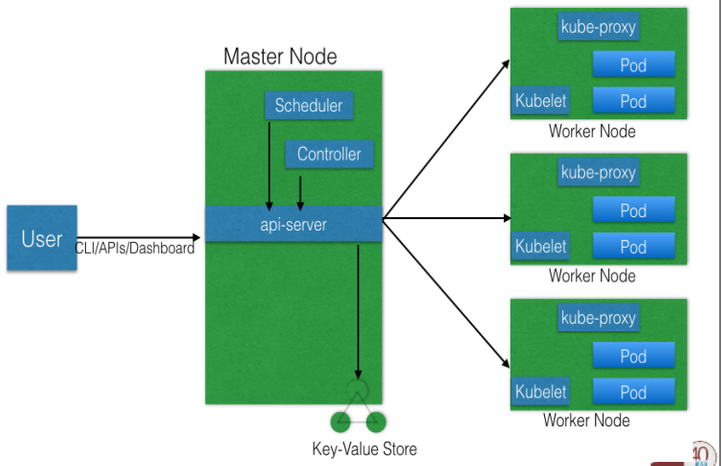
- For k8s, the Cloud Native Computing Foundation:
  - Provides a neutral home for the k8s trademark and enforces proper usage
  - Provides license scanning of core and vendored code
  - Offers legal guidance on patent and copyright issues
  - Creates open source curriculum, training, and certification
  - Manages a software conformance working group
  - Actively markets k8s
  - Hosts and funds developer marketing activities like K8Sport
  - Supports ad hoc activities, like offering a neutral k8s AMI in the AWS Marketplace
  - Funds conferences and meetup events.

Kubernetes architecture overview

In this section, we will explore the Kubernetes architecture, the different components of the Master and Worker nodes, the cluster state management with etcd and the network setup requirements. We will also talk about the network specification called Container Network Interface (CNI), which is used by Kubernetes.

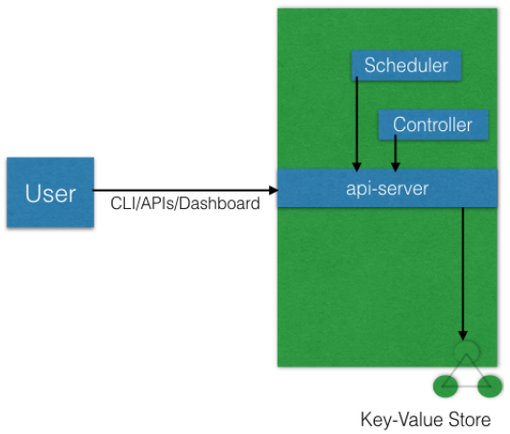
Kubernetes Architecture

- At a very high level, k8s has the following main components:
  - One or more Master Nodes
  - One or more Worker Nodes
  - Distributed key-value store, like etcd



Master Node

- Responsible for managing the k8s cluster, and it is the entry point for all administrative tasks
- We can communicate to the Master Node via the CLI, the GUI (Dashboard), or via APIs
- If we have more than one Master Node, they would be in a HA (High Availability) mode
- To manage the cluster state, k8s uses etcd, and all Master Nodes connect to it



For fault tolerance purposes, there can be more than one Master Node in the cluster. If we have more than one Master Node, they would be in a HA (High Availability) mode, and only one of them will be the leader, performing all the operations. The rest of the Master Nodes would be followers.
To manage the cluster state, Kubernetes uses etcd, and all Master Nodes connect to it. etcd is a distributed key-value store, which we will discuss in a little bit. The key-value store can be part of the Master Node. It can also be configured externally, in which case, the Master Nodes would connect to it.

Master Node Components I

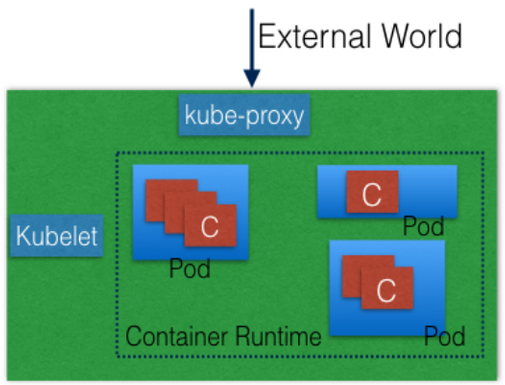
- **API Server**
  - All the administrative tasks are performed via the API Server within the Master Node
  - A user/operator sends REST commands to the API Server, which then validates and processes the requests
  - After executing the requests, the resulting state of the cluster is stored in the distributed key-value store.
- **Scheduler**
  - As the name suggests, the Scheduler schedules the work to different Worker Nodes
  - The Scheduler has the resource usage information for each Worker Node. It also knows about the constraints that users/operators may have set, such as scheduling work on a node that has the label disk==ssd set
  - Before scheduling the work, the Scheduler also takes into account the quality of the service requirements, data locality, affinity, anti-affinity, etc
  - The Scheduler schedules the work in terms of Pods and Services

Master Node Components II

- **Controller Manager**
  - Manages different non-terminating control loops, which regulate the state of the k8s cluster
  - Each one of these control loops knows about the desired state of the objects it manages, and watches their current state through the API Server
  - In a control loop, if the current state of the objects it manages does not meet the desired state, then the control loop takes corrective steps to make sure that the current state is the same as the desired state
- **etcd**
  - As discussed earlier, etcd is a distributed key-value store which is used to store the cluster state
  - It can be part of the k8s Master, or, it can be configured externally, in which case, Master Nodes would connect to it

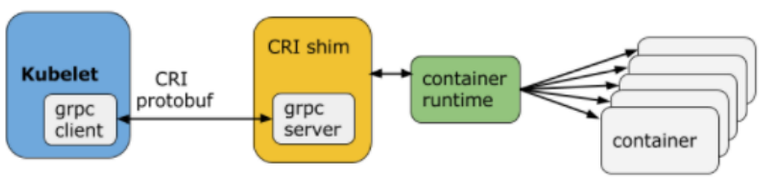
Worker Node

- A Worker Node is a machine (VM, physical server, etc.) which runs the applications using Pods and is controlled by the Master Node
- Pods are scheduled on the Worker Nodes, which have the necessary tools to run and connect them
- A Pod is the scheduling unit in k8s
- It is a logical collection of one or more containers which are always scheduled together
- Also, to access the applications from the external world, we connect to Worker Nodes and not to the Master Node/s



Worker Node Components I

- **Container Runtime**
  - To run containers, we need a Container Runtime on the Worker Node
  - By default, k8s is configured to run containers with Docker. It can also run containers using the rkt Container Runtime.
- **kubelet**
  - Is an agent which runs on each Worker Node and communicates with the Master Node
  - It receives the Pod definition via various means (primarily, through the API Server), and runs the containers associated with the Pod
  - It also makes sure the containers which are part of the Pods are healthy at all times.
  - The kubelet connects with the Container Runtimes to run containers



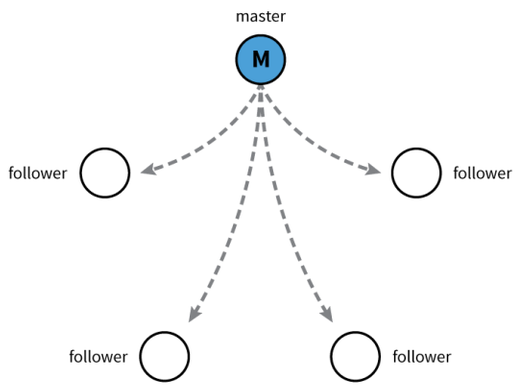
Currently, the kubelet and Container Runtimes are tightly coupled. There is work in progress for the Container Runtime Interface (CRI) to have a pluggable CRI in the near future.

Worker Node Components II

- **kube-proxy**
  - Instead of connecting directly to Pods to access the applications, we use a logical construct called a Service as a connection endpoint
  - A Service groups related Pods, which it load balances when accessed
  - **kube-proxy** is the network proxy which runs on each Worker Node and listens to the API Server for each Service endpoint creation/ deletion
  - For each Service endpoint, **kube-proxy** sets up the routes so that it can reach to it

State Management with etcd

- **etcd** is a distributed key-value store based on the Raft Consensus Algorithm
- Raft allows a collection of machines to work as a coherent group that can survive the failures of some of its members
- At any given time, one of the nodes in the group will be the Master, and the rest of them will be the Followers
- Any node can be treated as a Master



etcd is written in the Go programming language. In Kubernetes, besides storing the cluster state, etcd is also used to store configuration details such as subnets, ConfigMaps, Secrets, etc.

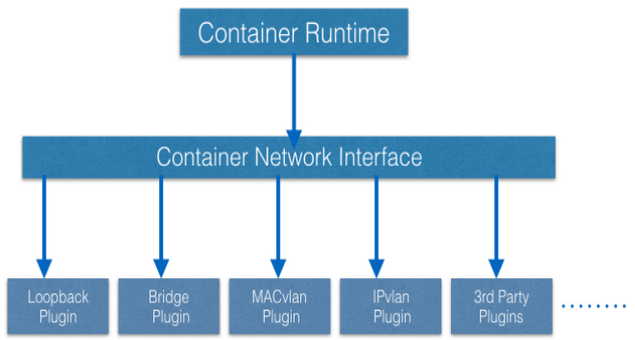
Network Setup Challenges

- To have a fully functional k8s cluster, we need to make sure of the following:
  - A unique IP is assigned to each Pod
  - Containers in a Pod can communicate to each other
  - The Pod is able to communicate with other Pods in the cluster
  - If configured, the application deployed inside a Pod is accessible from the external world
- All of the above are networking challenges which must be addressed before deploying the k8s cluster

AssertLab
Advanced Software and Systems Engineering Research Technologies

Assigning a Unique IP Address to Each Pod

- In k8s, each Pod gets a unique IP address
- For container networking, there are two primary specifications:
  - Container Network Model (CNM), proposed by Docker
  - Container Network Interface (CNI), proposed by CoreOS.
- k8s uses CNI to assign the IP address to each Pod.



The Container Runtime ffloads the IP assignment to CNI, which connects to the underlying configured plugin, like Bridge or MACvlan, to get the IP address. Once the IP address is given by the respective plugin, CNI forwards it back to the requested Container Runtime.

Container-to-Container Communication Inside a Pod

- With the help of the underlying Host OS, all of the Container Runtimes generally create an isolated network entity for each container that it starts
- On Linux, that entity is referred to as a Network Namespace
- These Network Namespaces can be shared across containers, or with the Host Operating System
- Inside a Pod, containers share the Network Namespaces, so that they can reach to each other via localhost

Pod-to-Pod Communication Across Nodes

- In a clustered environment, the Pods can be scheduled on any node
- We need to make sure that the Pods can communicate across the nodes, and all the nodes should be able to reach any Pod
- K8s also puts a condition that there shouldn't be any Network Address Translation (NAT) while doing the Pod-to-Pod communication across Hosts. We can achieve this via:
  - Routable Pods and nodes, using the underlying physical infrastructure, like Google Container Engine
  - Using Software Defined Networking, like Flannel, Weave, Calico, etc.
- For more details, you can take a look at the available k8s documentation

Communication Between the External World and Pods

- By exposing our services to the external world with kube-proxy, we can access our applications from outside the cluster
- We will have a complete section dedicated to this, so we will dive into this later

Installing Kubernetes

In this section, we will first discuss about the different configurations in which Kubernetes can be installed. We will then discuss about the infrastructure requirements to install Kubernetes, and we will also look at some of the tools which can help us with the installation.

Kubernetes Configuration

- **All-in-One Single-Node Installation**
  - With All-in-One, all the Master and Worker components are installed on a single node
  - This is very useful for learning, development, and testing
  - This type should not be used in production
  - **Minikube** is one such example, and we are going to explore it in future sections
- **Single-Node etcd, Single-Master, and Multi-Worker Installation**
  - In this setup, we will have a single Master Node, which will also run a single-node **etcd** instance
  - Multiple Worker Nodes are connected to the Master Node

Kubernetes can be installed using different configurations. The four major installation types are briefly presented

Kubernetes Configuration

- **Single-Node etcd, Multi-Master, and Multi-Worker Installation**
  - In this setup, we will have multiple Master Nodes, which will work in HA mode, but we will have a single-node etcd instance
  - Multiple Worker Nodes are connected to the Master Nodes
- **Multi-Node etcd, Multi-Master, and Multi-Worker Installation**
  - In this mode, **etcd** is configured in a clustered mode, outside the k8s cluster, and the Nodes connect to it
  - The Master Nodes are all configured in an HA mode, connecting to multiple Worker Nodes
  - This is the most advanced and recommended production setup

Infrastructure for Kubernetes Installation

- Once we decide on the installation type, we also need to make some infrastructure-related decisions, such as:
  - Should we set up k8s on bare-metal, public cloud, or private cloud?
  - Which underlying system should we use? Should we choose RHEL, CoreOS, CentOS, or something else?
  - Which networking solution should we use? And so on.
- The k8s documentation has details in regards to choosing the right solution

Localhost Installation

- There are a few localhost installation options available to deploy single- or multi-node k8s clusters on our workstation/laptop:
  - Minikube
  - Ubuntu on LXD.
- Minikube is the preferred and recommended way to create an all-in-one k8s setup
- We will be using it extensively in this lecture

On-Premise Installation

- **On-Premise VMs**
  - k8s can be installed on VMs created via Vagrant, VMware vSphere, KVM, etc
  - There are different tools available to automate the installation, like Ansible or kubeadm
- **On-Premise Bare Metal**
  - K8s can be installed on on-premise Bare Metal, on top of different Operating Systems, like RHEL, CoreOS, CentOS, Fedora, Ubuntu, etc
  - Most of the tools used to install VMs can be used with Bare Metal as well

Cloud Installation

- **Hosted Solutions**
  - With Hosted Solutions, any given software is completely managed by the provider
  - The user will just need to pay hosting and management charges
  - Some examples of vendors providing Hosted Solutions for k8s are listed below:
    - Google Container Engine (GKE)
    - Azure Container Service
    - OpenShift Dedicated
    - Platform9
    - IBM Bluemix Container Service

Cloud Installation

- **Turnkey Cloud Solutions**
  - With Turnkey Cloud Solutions, we can deploy a solution or software with just a few commands
  - For k8s, we have some Turnkey Cloud Solutions, with which k8s can be installed with just a few commands on an underlying IaaS platform, such as:
    - Google Compute Engine Amazon
    - AWS
    - Microsoft Azure
    - Tectonic by CoreOS

Cloud Installation

- **Bare Metal**
  - K8s can be installed on Bare Metal provided by di erent cloud providers

Kubernetes Installation Tools/Resources

- **kubeadm**
  - is a first-class citizen on the k8s ecosystem
  - It is a secure and recommended way to bootstrap the k8s cluster
  - It has a set of building blocks to setup the cluster, but it is easily extendable to add more functionality
  - Please note that **kubeadm** does not support the provisioning of machines.
- **Kubespray**
  - With Kubespray (formerly known as Kargo), we can install Highly Available k8s clusters on AWS, GCE, Azure, OpenStack, or Bare Metal
  - Kubespray is based on Ansible, and is available on most Linux distributions. It is a Kubernetes Incubator project.
- **Kops**
  - With Kops, we can create, destroy, upgrade, and maintain production-grade, highly-available k8s clusters from the command line. It can provision the machines as well
  - Currently, AWS is officially supported. Support for GCE and VMware vSphere are in alpha stage, and other platforms are planned for the future.

If the existing solutions and tools do not t your requirements, then you can always install Kubernetes from scratch.
It is worth checking out the Kubernetes The Hard Way GitHub project by Kelsey Hightower, which shares the manual steps involved in bootstrapping a Kubernetes cluster.

Setting Up a Single-Node Kubernetes Cluster with Minikube

As we mentioned in the previous section, Minikube is the easiest and most recommended way to run an All-in-One Kubernetes cluster locally. In this chapter, we will check out the requirements to install Minikube on our workstation, as well as the installation instructions to set it up on Linux, Mac and Windows.

Requirements for Running Minikube

- Minikube runs as a VM. Therefore, we need to make sure that we have the supported hardware and the hypervisor to create VMs
- kubectl: is a binary to access any k8s cluster
- On macOS: xhyve driver, VirtualBox or VMware Fusion hypervisors
- On Linux: VirtualBox or KVM hypervisors
- On Windows: VirtualBox or Hyper-V hypervisors
- VT-x/AMD-v virtualization must be enabled in BIOS
- Internet connection on first run

Generally, it is installed before starting minikube, but we can install it later, as well. If kubectl is not found while installing minikube, we will get a warning message, which can be safely ignored (just remember that we will have to install kubectl later). We will explore kubectl in sections.
In the section, we will use VirtualBox as hypervisor on all three operating systems - Linux, macOS, and Windows, to create the Minikube VM.

Installing Minikube on Linux

Installing Minikube on macOS

Installing Minikube on Windows

Accessing Minikube

In this section, we will study the different access methods to any Kubernetes cluster. We will use kubectl to access Minikube via CLI, the Kubernetes dashboard to access it via GUI, and the curl command, with the right credentials to access it via APIs.

Accessing Minikube

- **Command Line Interface (CLI)**
  - kubectl is the CLI tool to manage the k8s cluster resources and applications
- **Graphical User Interface (GUI)**
  - K8s dashboard provides the GUI to interact with its resources and containerized applications.
- **APIs**
  - As we know, k8s has the API Server, and operators/users connect to it from the external world to interact with the cluster
  - Using both CLI and GUI, we can connect to the API Server on the Master Node to perform different operations
  - We can directly connect to the API Server using its API endpoints and send commands to it, as long as we can access the Master Node and have the right credentials

The methods we just outlined are applicable to all Kubernetes clusters. Next, we will see how we can access the minikube environment we set up in the previous section.

kubectl

- Is generally installed before installing minikube, but we can also install it later
- There are different methods that can be used to install kubectl, which are mentioned in the k8s Documentation.
- Next, we will look at the steps to install kubectl on Linux, macOS, and Windows systems.

Installing kubectl on Linux

- Download the latest stable **kubectl** binary
- `$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl`
- Make the **kubectl** binary executable
- `$ chmod +x ./kubectl`
- Move the **kubectl** binary to the **PATH**
- `$ sudo mv ./kubectl /usr/local/bin/kubectl`

AssertLab
Advanced Software and Systems Engineering Research Technologies

Installing kubectl on macOS

- To manually install **kubectl** on macOS, follow the instructions below:
  - Download the latest stable **kubectl** binary
    - `$ curl-LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl`
  - Make the **kubectl** binary executable
    - `$ chmod +x ./kubectl`
  - Move the **kubectl** binary to the **PATH**
    - `$ sudo mv ./kubectl /usr/local/bin/kubectl`
- To install **kubectl** on macOS using the Homebrew package manager, do:
  - `$ brew install kubectl`

AssertLab
Advanced Software and Systems Engineering Research Technologies

Installing kubectl on Windows

- To install **kubectl** on Windows, follow the steps below:
  - Get the latest **kubectl** release from here
  - Depending on the latest release, download the **kubectl** binary. In the example below, we are downloading the v1.6.3 release
    - `$ curl -LO https://storage.googleapis.com/kubernetesrelease/release/v1.6.3/bin/windows/amd64/kubectl.exe`
- Once downloaded, move the kubectl binary to the PATH

kubectl Configuration File

- To connect to the k8s cluster, **kubectl** needs the Master Node endpoint and the credentials to connect to it
- While starting minikube, the startup process creates, by default, a configuration file, `config`, inside the `.kube` directory, which resides in user's home directory
- That configuration file has all the connection details
- By default, the **kubectl** binary accesses this file to find the Master Node's connection endpoint, along with the credentials

```
$ kubectl config view
apiVersion: v1
clusters:
 - cluster:
    certificate-authority: /Users/nkhare/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
 - context:
    cluster: minikube
    user: minikube
  name: minikube
currentcontext: minikube
kind: Config
preferences: {}
users:
 - name: minikube
  user:
    clientcertificate: /Users/nkhare/.minikube/apiserver.crt
    clientkey: /Users/nkhare/.minikube/apiserver.key
```

kubectl Configuration File

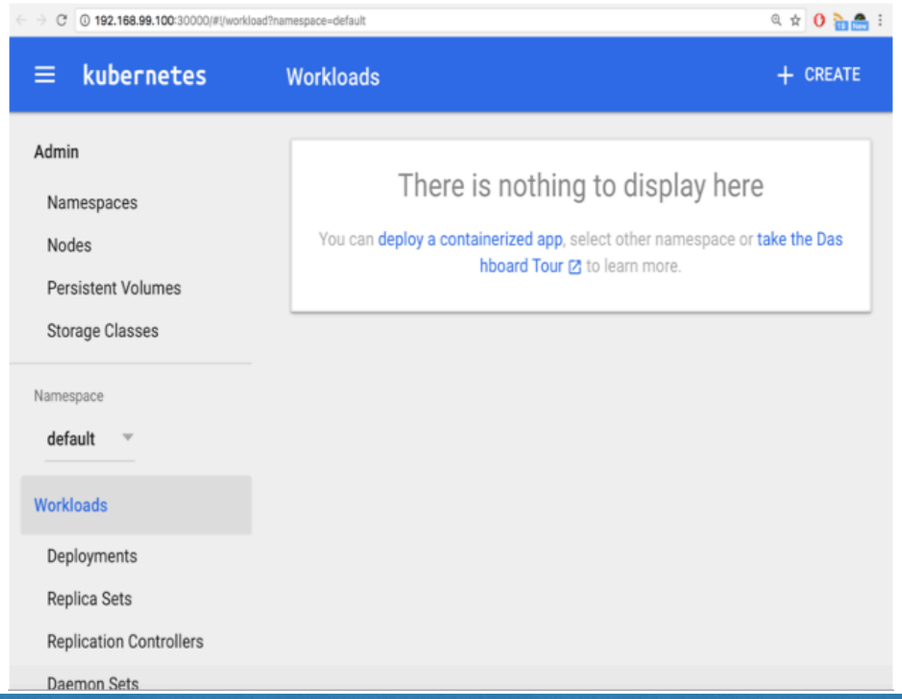
- Once **kubectl** is installed, we can get information about the minikube cluster with the `kubectl cluster-info` command:

```
$ kubectl cluster-info

Kubernetes master is running at https://
192.168.99.100:8443

To further debug and diagnose cluster problems,
use ‘kubectl cluster-info dump'.
```

- You can find more details about the kubectl command line options here

Using the 'minikube dashboard' Command

- As mentioned earlier, the k8s Dashboard provides the user interface for the k8s cluster
- To access the Dashboard of Minikube, we can use minikube dashboard, which would open a new tab on our web browser, displaying the k8s dashboard:
  - `$ minikube dashboard`

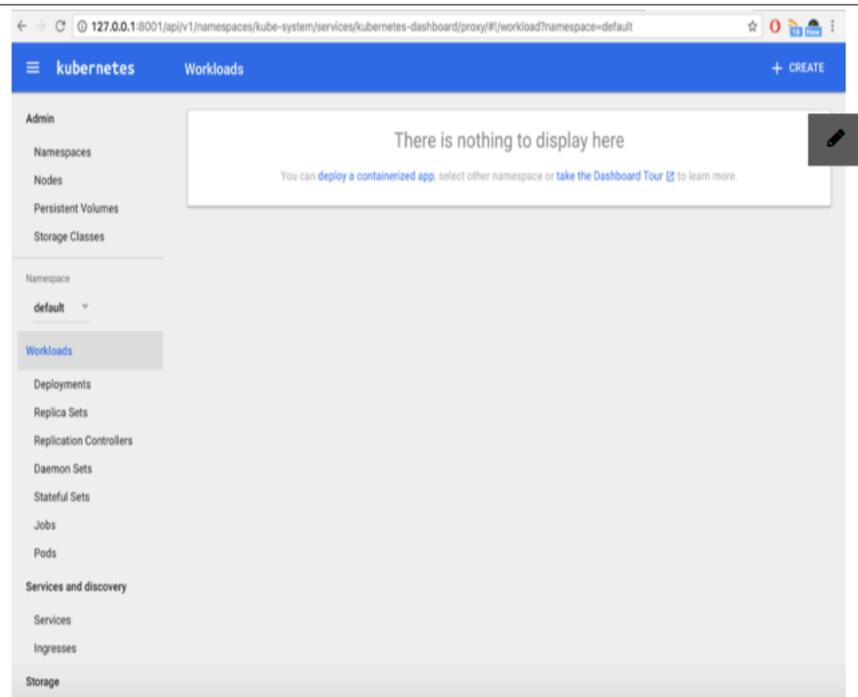

192.168.99.100:30000/#!/workload?namespace=default
kubernetes
Workloads
+ CREATE
Admin
Namespaces
Nodes
Persistent Volumes
Storage Classes
Namespace
default
Workloads
Deployments
Replica Sets
Replication Controllers
Daemon Sets
There is nothing to display here
You can deploy a containerized app, select other namespace or take the Dashboard Tour to learn more.

Using the 'kubectl proxy' Command

- Using the `kubectl proxy` command, **kubectl** would authenticate with the API Server on the Master Node and would make the dashboard available on `http://localhost:8001/ui`

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

- After running the above command, we can access the dashboard at `http://127.0.0.1:8001/ui`



APIs - with 'kubectl proxy'

- When kubectl proxy is configured, we can send requests to localhost on the proxy port
- With the above curl request, we requested all the API endpoints from the API Server.

```
$ curl http://localhost:8001/
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/apps",
    ......
    ......
    "/logs",
    "/metrics",
    "/swaggerapi/",
    "/ui/",
    "/version"
  ]
}%
```

APIs - without 'kubectl proxy'

- Without `kubectl proxy` configured, we can get the **Bearer Token** using **kubectl**, and then send it with the API request
- A **Bearer Token** is an **access token** which is generated by the authentication server (the API server on the Master Node) and given back to the client
- Using that token, the client can connect back to the k8s API server without providing further authentication details, and then, access resources

APIs - without 'kubectl proxy'

- Get the token
  - `$ TOKEN=$(kubectl describe secret $ (kubectl get secrets | grep default | cut -f1 -d ' ') | grep E '^token' | cut -f2 -d':' | tr -d '\t')`
- Get the API Server endpoint
  - `$ APISERVER=$(kubectl config view | grep https | cut -f 2- -d ":" | tr -d " ")`

Access the API Server using the curl command

```
$ curl $APISERVER header "Authorization: Bearer $TOKEN"
   insecure
   {
    "paths": [
      "/api",
      "/api/v1",
      "/apis",
      "/apis/apps",
      ......
      ......
      "/logs",
      "/metrics",
      "/swaggerapi/",
      "/ui/",
      "/version"
    ]
}%
```

AssertLab
Advanced Software and Systems Engineering Research Technologies

Kubernetes Building Blocks

In this section we will explore the Kubernetes Object Model and discuss some of its building blocks, such as Pods, ReplicaSets, Deployments, Namespaces, etc. We will also discuss the role of Labels and Selectors when it comes to grouping objects together.

Kubernetes Object Model

- K8s has a very rich object model, with which it represents different persistent entities in the k8s cluster. Those entities describe:
  - What containerized applications we are running and on which node
  - Application resource consumption
  - Different policies attached to applications, like restart/upgrade policies, fault tolerance, etc.
- With each object, we declare our intent or desired state using the spec field
- The k8s system manages the status field for objects, in which it records the actual state of the object
- At any given point in time, the k8s Control Plane tries to match the object's actual state to the object's desired state
- Examples: Pods, Deployments, ReplicaSets, etc.

Kubernetes Object Model

- To create an object, we need to provide the spec field to the k8s API Server
- The `spec` field describes the desired state, along with some basic information, like the name
- The API request to create the object must have the `spec` field, as well as other details, in a JSON format
- Most often, we provide an object's definition in a `.yaml` file, which is converted by **kubectl** in a JSON payload and sent to the API Server

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Kubernetes Object Model

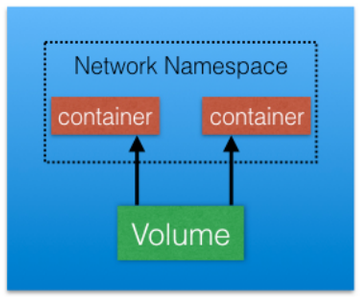
- With the `apiVersion` field in the example above, we mention the API endpoint on the API Server which we want to connect to
- With the `kind` field, we mention the object type - in our case, we have `Deployment`
- With the `metadata` field, we attach the basic information to objects, like the name
- You may have noticed that in above we have two `spec` fields (`spec` and `spec.template.spec`)
  - With `spec`, we define the desired state of the deployment
  - In `spec.template.spec`, we define the desired state of the Pod - here, our Pod would be created using `nginx:1.7.9`

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```


In our example, we want to make sure that, at any point in time, at least 3 Pods are running, which are created using the Pod template defined in spec.template. Once the object is created, the Kubernetes system attaches the status field to the object.

Pods

- A Pod is the smallest and simplest k8s object
- It is the unit of deployment in k8s, which represents a single instance of the application
- A Pod is a logical collection of one or more containers, which:
  - Are scheduled together on the same host
  - Share the same network namespace
  - Mount the same external storage (Volumes)
- Pods are ephemeral in nature, and they do not have the capability to self-heal by themselves



That is why we use them with controllers, which can handle a Pod's replication, fault tolerance, self-heal, etc. Examples of controllers are Deployments, ReplicaSets, ReplicationControllers, etc. We attach the Pod's specification to other objects using Pod Templates, as we have seen in the previous section.

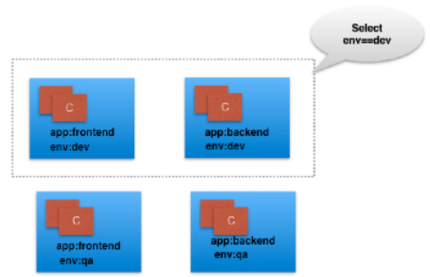
Labels

- Labels are key-value pairs that can be attached to any k8s objects (e.g. Pods)
- Labels are used to organize and select a subset of objects, based on the requirements in place
- Many objects can have the same label(s)
- Labels do not provide uniqueness to objects

In the image above, we have used two labels: app and env. Based on our requirements, we have given different values to our four Pods

Label Selectors

- **Equality-Based Selectors**
  - Allow filtering of objects based on label keys and values
  - With this type of Selectors, we can use the `=`, `==`, or `!=` operators
  - For example, with `env==dev` we are selecting the objects where the `env` label is set to `dev`
- **Set-Based Selectors**
  - Allow filtering of objects based on a set of values
  - With this type of Selectors, we can use the `in`, `notin`, and `exist` operators
  - For example, with `env in (dev,qa)`, we are selecting objects where the `env` label is set to `dev` or `qa`



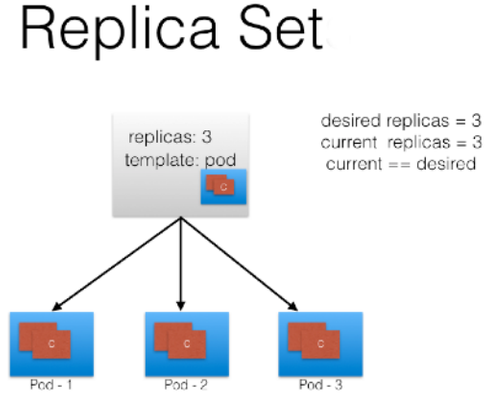
With Label Selectors, we can select a subset of objects. Kubernetes supports two types of Selectors.

Replication Controllers

- A ReplicationController (`rc`) is a controller that is part of the Master Node's Controller Manager
- It makes sure the specified number of replicas for a Pod is running at any given point in time
- If there are more Pods than the desired count, the **ReplicationController** would kill the extra Pods, and, if there are less Pods, then the **ReplicationController** would create more Pods to match the desired count
- Generally, we don't deploy a Pod independently, as it would not be able to re-start itself, if something goes wrong
- We always use controllers like **ReplicationController** to create and manage Pods

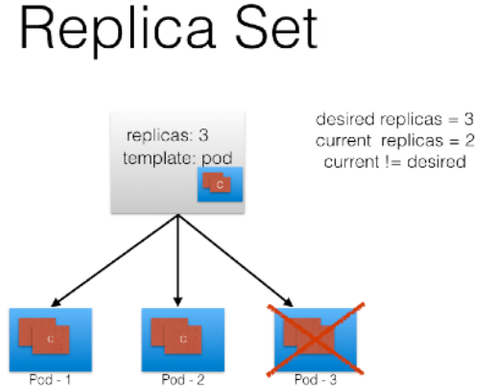
ReplicaSets I

- A ReplicaSet (`rs`) is the next-generation ReplicationController
- ReplicaSets support both equality- and set-based Selectors, whereas ReplicationControllers only support equality-based Selectors
- Next, you can see a graphical representation of a ReplicaSet, where we have set the replica count to 3 for a Pod
- Now, let's suppose that one Pod dies, and our current state is not matching the desired state anymore



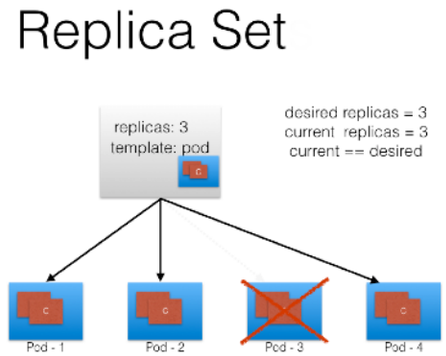
ReplicaSets I

- A ReplicaSet (`rs`) is the next-generation ReplicationController
- ReplicaSets support both equality- and set-based Selectors, whereas ReplicationControllers only support equality-based Selectors
- Next, you can see a graphical representation of a ReplicaSet, where we have set the replica count to 3 for a Pod
- Now, let's suppose that one Pod dies, and our current state is not matching the desired state anymore



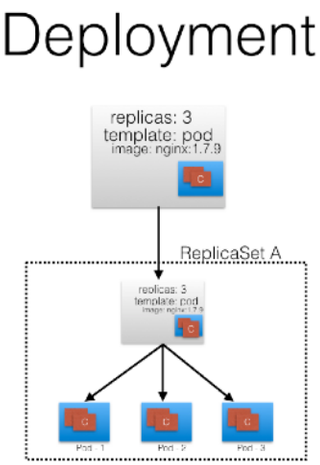
ReplicaSets II

- The ReplicaSet will detect that the current state is no longer matching the desired state
- So, in our given scenario, the ReplicaSet will create one more Pod, thus ensuring that the current state matches the desired state
- ReplicaSets can be used independently, but they are mostly used by Deployments to orchestrate the Pod creation, deletion, and updates
- A Deployment automatically creates the ReplicaSets, and we do not have to worry about managing them.



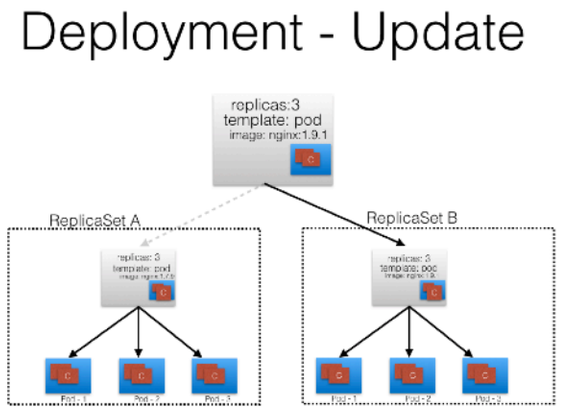
Deployments

- Deployment objects provide declarative updates to Pods and ReplicaSets
- The DeploymentController is part of the Master Node's Controller Manager, and it makes sure that the current state always matches the desired state
- In the following example, we have a **Deployment** which creates a `ReplicaSet A`
  - `ReplicaSet A` then creates **3 Pods**
  - In each Pod, one of the containers uses the `nginx:1.7.9` image



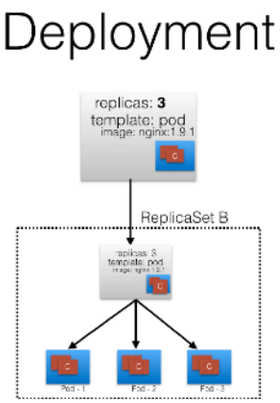
Deployments

- Now, in the Deployment, we change the Pod's template and we update the image for the `nginx` container from `nginx:1.7.9` to `nginx:1.9.1`
- As have modified the Pod's template, a new `ReplicaSet B` gets created
- This process is referred to as a **Deployment rollout**
  - A rollout is only triggered when we update the Pod's template for a deployment
  - Operations like **scaling the deployment** do not trigger the deployment



Deployments

- Once `ReplicaSet B` is ready, the Deployment starts pointing to it
- On top of ReplicaSets, Deployments provide features like Deployment recording, with which, if something goes wrong, we can rollback to a previously known state



Namespaces

- If we have numerous users whom we would like to organize into teams/projects, we can partition the k8s cluster into sub-clusters using Namespaces
- The names of the resources/objects created inside a Namespace are unique, but not across Namespaces
- To list all the Namespaces, we can run the following command:

```
$ kubectl get namespaces
NAME          STATUS       AGE
default       Active       11h
kube-public   Active       11h
kube-system   Active       11h
```

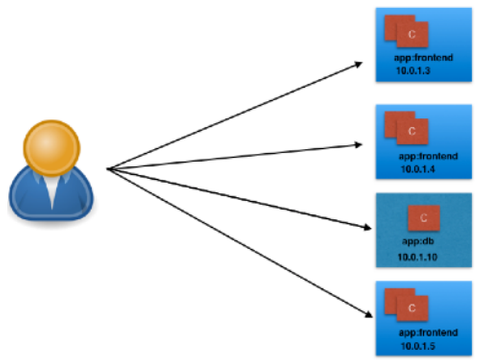

Generally, Kubernetes creates two default namespaces: kube-system and default. The kube-system namespace contains the objects created by the Kubernetes system. The default namespace contains the objects which belong to any other namespace. By default, we connect to the default Namespace. kube-public is a special namespace, which is readable by all users and used for special purposes, like bootstrapping a cluster.
Using Resource Quotas, we can divide the cluster resources within Namespaces.

Services

In this section, we will learn about Services, using which we can group Pods to provide common access points from the external world. We will learn about the kube-proxy daemon, which runs on each Worker Node to provide access to Services. We will also discuss about Service Discovery and Service Types, which decide the access scope of a service.

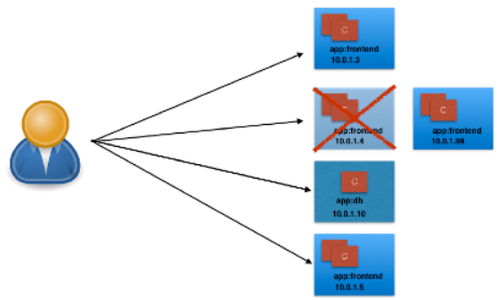
Connecting Users to Pods

- To access the application, a user/ client needs to connect to the Pods
- As Pods are ephemeral in nature, resources like IP addresses allocated to it cannot be static
- Pods could die abruptly or be rescheduled based on existing requirements
- Unexpectedly, the Pod to which the user/client is connected dies, and a new Pod is created by the controller



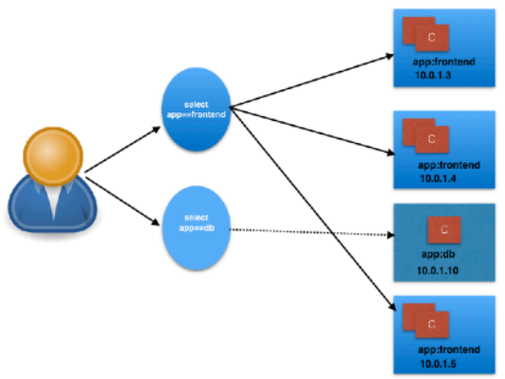
Connecting Users to Pods

- The new Pod will have a new IP address, which will not be known automatically to the user/client of the earlier Pod
- To overcome this situation, k8s provides a higher-level abstraction called Service, which logically groups Pods and a policy to access them
- This grouping is achieved via Labels and Selectors, which we talked about in the previous chapter.



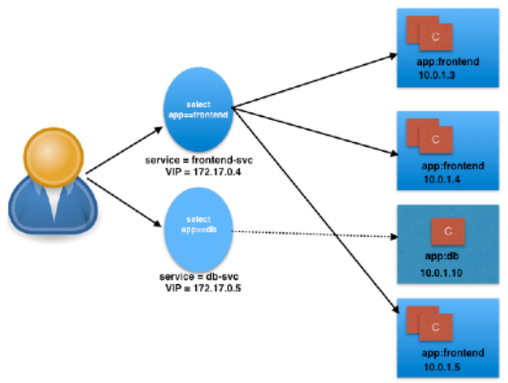
Services

- Here we have used the app keyword as a Label, and frontend and db as values for different Pods
- Using Selectors (`app==frontend` and `app==db`), we can group them into two logical groups: one with 3 Pods, and one with just one Pod



Services

- We can assign a name to the logical grouping, referred to as a **service name**
- In our example, we have created two Services, `frontend-svc` and `db-svc`, and they have the `app==frontend` and the `app==db` Selectors, respectively



AssertLab
Advanced Software and Systems Engineering Research Technologies

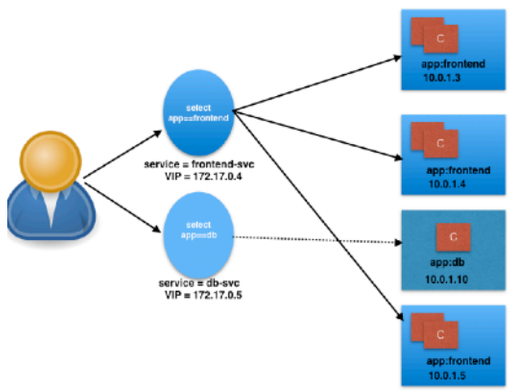
Service Object Example

- We are creating a `frontend-svc` Service by selecting all the Pods that have the Label `app` set to the `frontend`
- By default, each Service also gets an IP address, which is routable only inside the cluster
- In our case, we have `172.17.0.4` and `172.17.0.5` IP addresses for our `frontend-svc` and `db-svc` Services, respectively
- The IP address attached to each Service is also known as the ClusterIP for that Service

```
kind: Service
apiVersion: v1
metadata:
  name: frontend-svc
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
```

Service Object Example

- The user/client now connects to a Service via the IP address, which forwards the traffic to one of the Pods attached to it
- A Service does the load balancing while selecting the Pods for forwarding the data/traffic
- While forwarding the traffic from the Service, we can select the target port on the Pod
- A tuple of Pods, IP addresses, along with the targetPort is referred to as a Service Endpoint

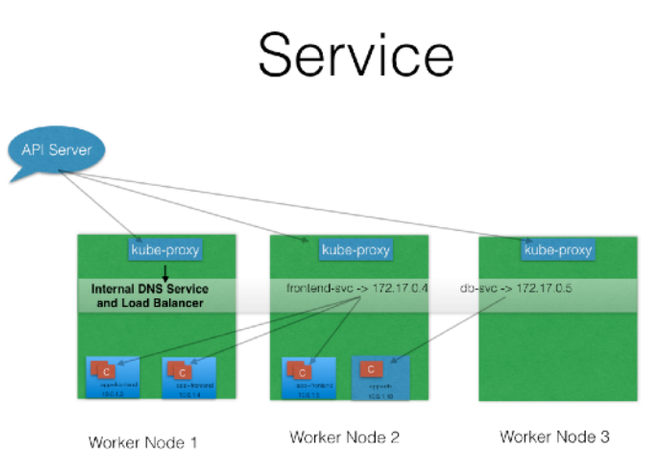


In our example, for frontend-svc, we will receive requests from the user/client on Port 80. We will then forward these requests to one of the attached Pods on Port 5000. If the target port is not defined explicitly, then traffic will be forwarded to Pods on the Port on which the Service receives traffic.

In our case, frontend-svc has 3 Endpoints: 10.0.1.3:5000, 10.0.1.4:5000, and 10.0.1.5:5000.

kube-proxy

- All of the Worker Nodes run a daemon called kube-proxy, which watches the API Server on the Master Node for the addition and removal of Services and endpoints
- For each new Service, on each node, **kube-proxy** configures the IPtables rules to capture the traffic for its ClusterIP and forwards it to one of the endpoints
- When the Service is removed, **kube-proxy** removes the IPtables rules on all nodes as well



Service Discovery

- **Environment Variables**
  - As soon as the Pod starts on any Worker Node, the `kubelet` daemon running on that node adds a set of environment variables in the Pod for all active Services
  - For example, if we have an active Service called redis-master, which exposes port 6379, and its ClusterIP is 172.17.0.6, then, on a newly created Pod:

```
REDIS_MASTER_SERVICE_HOST=172.17.0.6
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://172.17.0.6:6379
REDIS_MASTER_PORT_6379_TCP=tcp://172.17.0.6:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=172.17.0.6
```

  - With this solution, we need to be careful while ordering our Services, as the Pods will not have the environment variables set for Services which are created after the Pods are created

As Services are the primary mode of communication in Kubernetes, we need a way to discover them at runtime. Kubernetes supports two methods of discovering a Service

Service Discovery

- **DNS**
  - K8s has an add-on for DNS, which creates a DNS record for each Service and its format is like my-svc.my-`namespace.svc.cluster.local`
  - Services within the same namespace can reach to other services with just their name
  - For example, if we add a Service redis-master in the `my-ns` Namespace, then all the Pods in the same Namespace can reach to the `redis` Service just by using its name, `redis-master`
  - Pods from other Namespaces can reach the Service by adding the respective Namespace as a suffix, like redis-master.my-`ns`

This is the most common and highly recommended solution

For example, in the previous section's image, we have seen that an internal DNS is configured, which maps our services frontend-svc and db-svc to 172.17.0.4 and 172.17.0.5, respectively

Service Type

- While defining a Service, we can also choose its access scope. We can decide whether the Service:
  - Is only accessible within the cluster
  - Is accessible from within the cluster and the external world
  - Maps to an external entity which resides outside the cluster.
- Access scope is decided by ServiceType, which can be mentioned when creating the Service

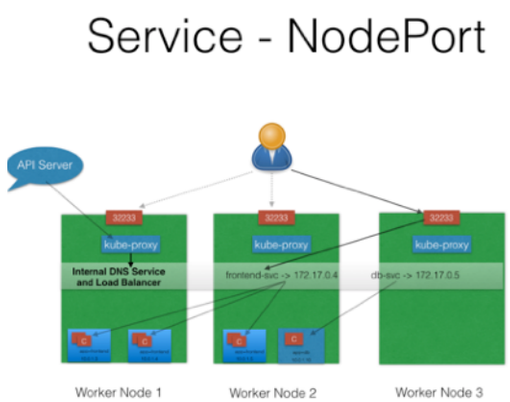
ServiceType - ClusterIP and NodePort

- **ClusterIP** is the default *ServiceType*
  - A Service gets its Virtual IP address using the ClusterIP
  - That IP address is used for communicating with the Service and is accessible only within the cluster.
- With the **NodePort** *ServiceType*, in addition to creating a ClusterIP, a port from the range `30000-32767` is mapped to the respective service, from all the Worker Nodes
  - For example, if the mapped NodePort is `32233` for the service `frontend-svc`, then, if we connect to any Worker Node on port `32233`, the node would redirect all the traffic to the assigned ClusterIP - `172.17.0.4`

By default, while exposing a NodePort, a random port is automatically selected by the Kubernetes Master from the port range 30000-32767. If we don't want to assign a dynamic port value for NodePort, then, while creating the Service, we can also give a port number from the earlier specific range.

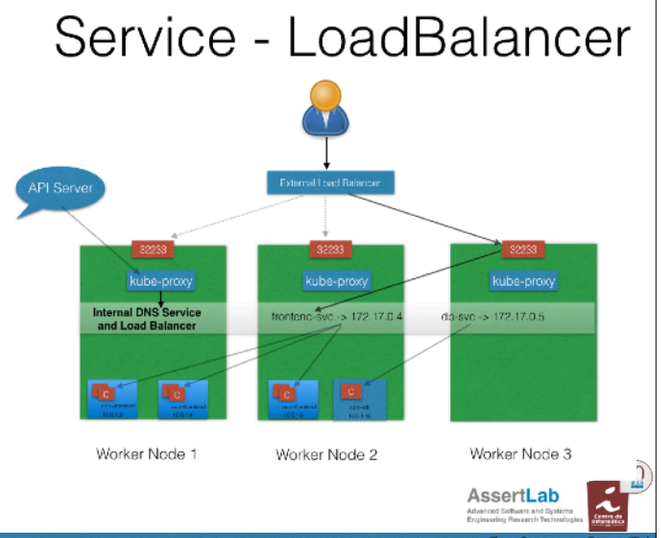
Service NodePort

- The **NodePort** *ServiceType* is useful when we want to make our services accessible from the external world
- The end-user connects to the Worker Nodes on the specified port, which forwards the traffic to the applications running inside the cluster
- To access the application from the external world, administrators can configure a reverse proxy outside the k8s cluster and map the specific endpoint to the respective port on the Worker Nodes



ServiceType - LoadBalancer

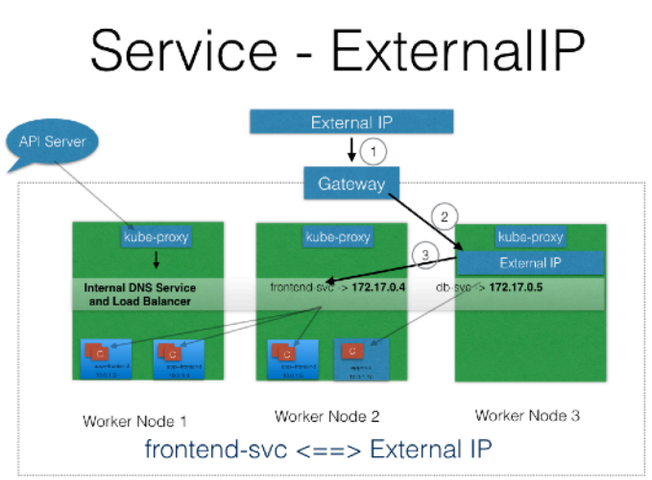
- NodePort and ClusterIP Services are automatically created, and the external load balancer will route to them
- The Services are exposed at a static port on each Worker Node
- The Service is exposed externally using the underlying Cloud provider's load balancer feature



The LoadBalancer ServiceType will only work if the underlying infrastructure supports the automatic creation of Load Balancers and have the respective support in Kubernetes, as is the case with the Google Cloud Platform and AWS

ServiceType - ExternalIP

- A Service can be mapped to an ExternalIP address if it can route to one or more of the Worker Nodes
- Traffic that is ingressed into the cluster with the ExternalIP (as destination IP) on the Service port, gets routed to one of the the Service endpoints



Please note that ExternalIPs are not managed by Kubernetes. The cluster administrators has configured the routing to map the ExternalIP address to one of the nodes.

ServiceType - ExternalName

- **ExternalName** is a special *ServiceType*, that has no Selectors and does not define any endpoints
- When accessed within the cluster, it returns a `CNAME` record of an externally configured service
- The primary use case of this *ServiceType* is to make externally configured services like `my-database.example.com` available inside the cluster, using just the name, like `my-database`, to other services inside the same Namespace

Deploying a Stand-Alone Application

In earlier sections, we looked at the Kubernetes architecture and its building blocks (objects). We also saw how we can deploy and access an All-in-One Kubernetes cluster using minikube. In this chapter, we will learn to deploy an application using the GUI or CLI. We will also expose an application with NodePort, and access it from the external world.

Deploying an Application Using the Minikube GUI I

- **Make sure that minikube is running**

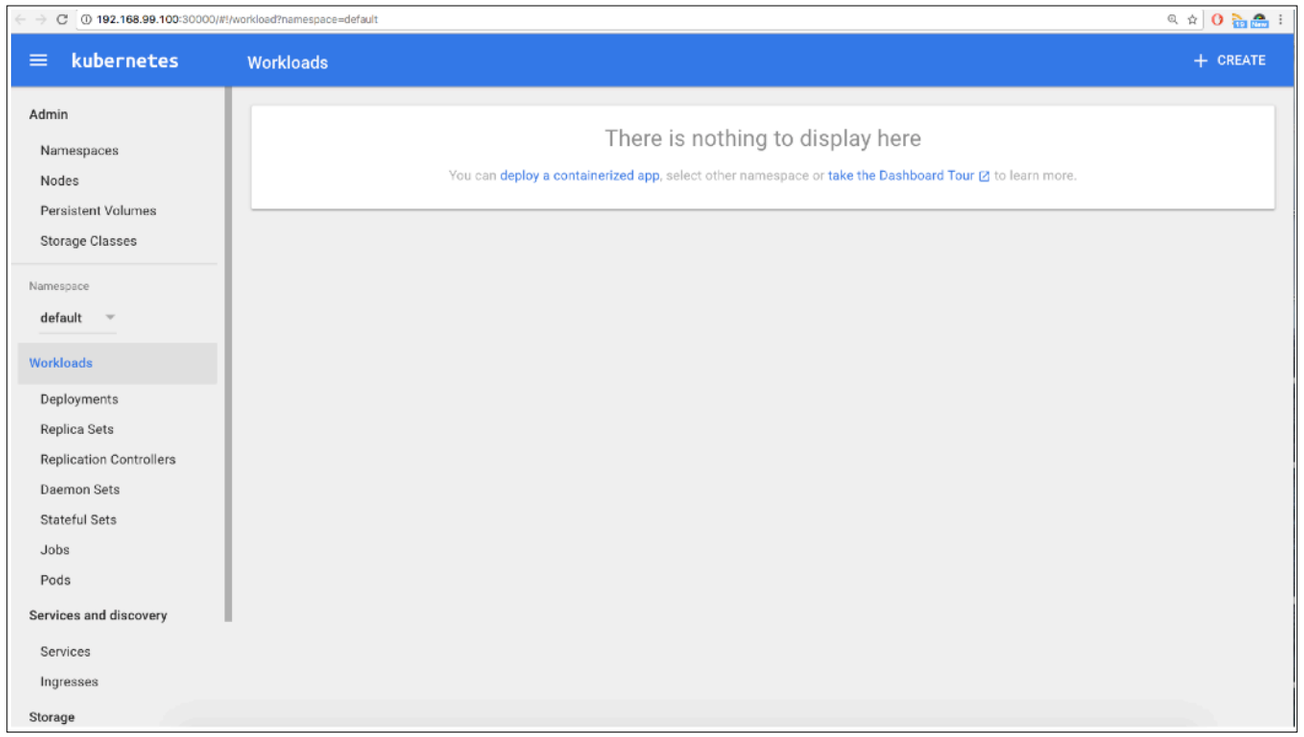
```
$ minikube status
minikubeVM: Running
localkube: Running
```

- **Start the minikube Dashboard**

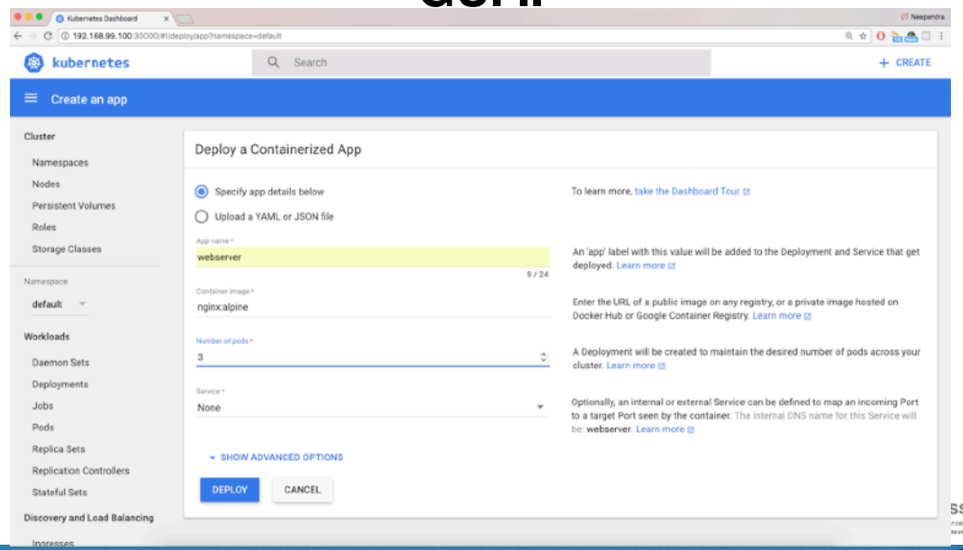
```
$ minikube dashboard
```

- Running this command will open up a browser with the Kubernetes GUI, which we can use to manage containerized applications

In the next few slides, we will learn how to deploy an nginx webserver using the nginx:alpine Docker image.
By default, the Dashboard is connected to the default Namespace. So, all the operations that we will do in this chapter will be performed inside the default Namespace


192.168.99.100:30000/#!/workload?namespace=default
kubernetes
Workloads
CREATE
Admin
Namespaces
Nodes
Persistent Volumes
Storage Classes
Namespace
default
Workloads
Deployments
Replica Sets
Replication Controllers
Daemon Sets
Stateful Sets
Jobs
Pods
Services and discovery
Services
Ingresses
Storage
There is nothing to display here
You can deploy a containerized app, select other namespace or take the Dashboard Tour to learn more.

Deploying an Application Using the Minikube GUI II



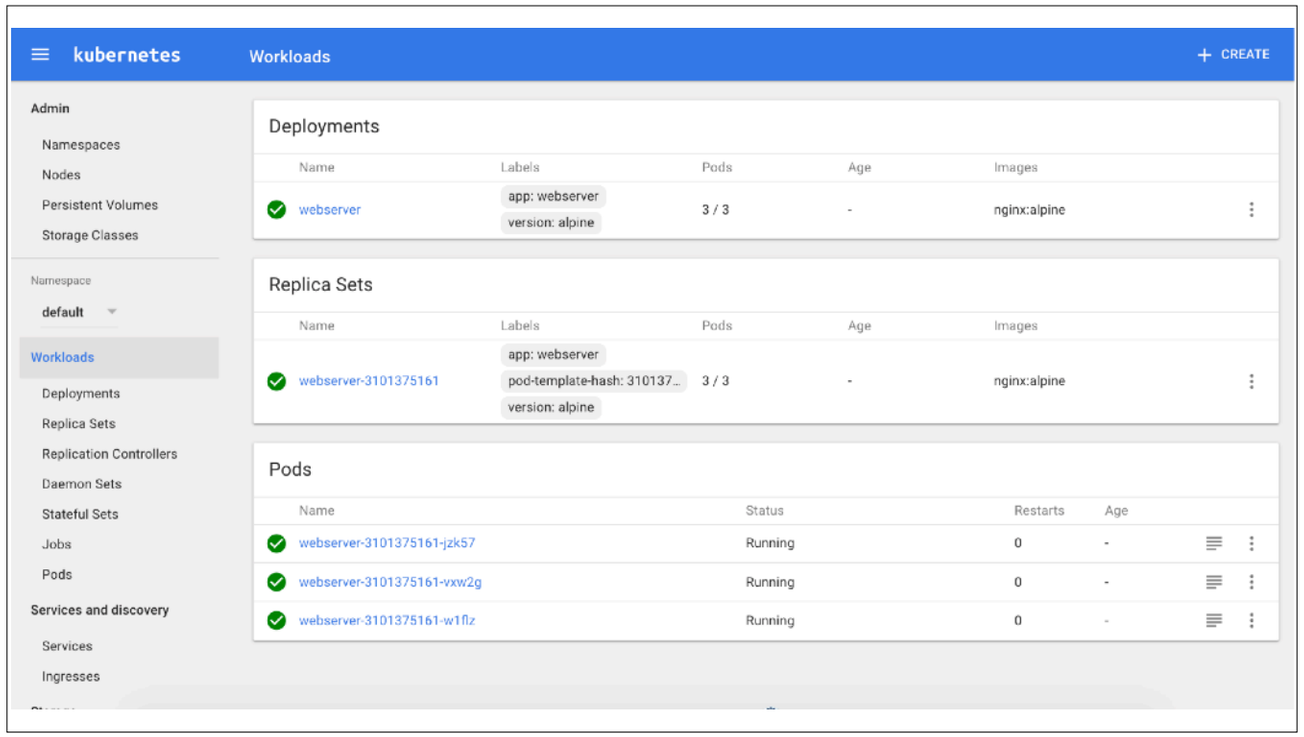
From the Dashboard, click on Deploy a Containerized App, which will open an interface like the one

Deploying an Application Using the Minikube GUI II

- We can either provide the application details here, or we can upload a YAML file with our Deployment details. As shown, we are providing the following application details:
  - The application name is `webserver`
  - The Docker image to use is `nginx:alpine`, where `alpine` is the image tag
  - The replica count, or the number of Pods, is 3
  - No Service, as we will be creating it later

Deploying an Application Using the Minikube GUI II

- If we click on *SHOW ADVANCED OPTIONS*, we can specify options such as Labels, Environment Variables, etc
  - By default, the Label is set to the application name
  - In our example, an `app:webserver` Label is set to different objects created by this Deployment.
- By clicking on the *DEPLOY* button, we can trigger the deployment.
  - As expected, the Deployment `webserver` will create a ReplicaSet (`webserver-1529352408`), which will eventually create three Pods (`webserver-1529352408-xxxx`)


kubernetes
Workloads
CREATE
Admin
Namespaces
Nodes
Persistent Volumes
Storage Classes
Namespace
default
Workloads
Deployments
Replica Sets
Replication Controllers
Daemon Sets
Stateful Sets
Jobs
Pods
Services and discovery
Services
Ingresses
Deployments
Name
Labels
Pods
Age
Images
webserver
app: webserver
version: alpine
3 / 3
-
nginx:alpine
Replica Sets
Name
Labels
Pods
Age
Images
webserver-3101375161
app: webserver
pod-template-hash: 310137...
version: alpine
3 / 3
-
nginx:alpine
Pods
Name
Status
Restarts
Age
webserver-3101375161-jzk57
Running
0
-
webserver-3101375161-vxw2g
Running
0
-
webserver-3101375161-w1flz
Running
0
-

Check out Deployments, ReplicaSets, and Pods from the CLI

- List the Deployment in the given Namespace:

```
$ kubectl get deployments
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
webserver   3         3         3            3           3m
```

- List the ReplicaSets in the given Namespace:

```
$ kubectl get replicasets
NAME                   DESIRED   CURRENT   READY   AGE
webserver-3101375161   3         3         3       4m
```

- List the Pods in the given Namespace:

```
$ kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
webserver-3101375161-jzk57     1/1     Running   0          4m
webserver-3101375161-vxw2g     1/1     Running   0          4m
webserver-31013755161-w1flz    1/1     Running   0          4m
```

Exploring Labels and Selectors I

- Earlier, we have seen that Labels and Selectors play an important role in grouping a subset of objects on which we can perform operations
- Next, we will take a closer look at them

Look at a Pod's details

- As you were able to see, with kubectl's describe subcommand we can get all the details about a Pod

```
$ kubectl describe pod webserver-3101375161-jzk57
Name:         webserver-3101375161-jzk57
Namespace:    default
Node:         minikube/192.168.99.100
Start Time:   Mon, 29 May 2017 12:58:34 +0530
Labels:       app=webserver
              pod-template-hash=3101375161
              version=alpine
Annotations: kubernetes.io/created-
by={"kind":"SerializedReference","apiVersion":"v1","reference":
{"kind":"ReplicaSet","namespace":"default","name":"webserver-
3101375161","uid":"6c85e06b-4440-11e7-9c3b-080027a4605...
Status:       Running
IP:           172.17.0.4
........
........
```


We can look at an object's details using kubectl's describe subcommand. In the following example, you can see a Pod's description

List the Pods, along with their attached Labels

```
$ kubectl get pods -L app,label2
NAME                          READY   STATUS    RESTARTS   AGE   APP
LABEL2
webserver-3101375161-jzk57    1/1     Running   0          12m   webserver
<none>
webserver-3101375161-vxw2g    1/1     Running   0          12m   webserver
<none>
webserver-3101375161-w1flz    1/1     Running   0          12m   webserver
<none>
```

- All of the Pods are listed, as each Pod has the Label app, whose value is set to webserver
- We can see that from the APP column
- As none of the Pods has the label2 Label, the value <none> is listed under the LABEL2 column

With the -L option to the kubectl get pods command, we can add additional columns in the output to list Pods with their attached Labels and their values. In the following example, we are listing Pods with the Labels app and label2

Select the Pods with a given Label

```
$ kubectl get pods -l app=webserver
NAME                         READY   STATUS    RESTARTS   AGE
webserver-3101375161-jzk57   1/1     Running   0          16m
webserver-3101375161-vxw2g   1/1     Running   0          16m
webserver-3101375161-w1flz   1/1     Running   0          16m
```


To use a Selector with the kubectl get pods command, we can use the -l option. In the following example, we are selecting all the Pods that have the app Label's value set to webserver.
In the example above, we listed all the Pods we created, as all of them have the app Label, whose value is set to webserver

Delete the Deployment we created earlier

- We can delete any object using the kubectl delete command

```
$ kubectl delete deployments webserver
deployment "webserver" deleted
```

- Deleting a Deployment also deletes the ReplicaSets and the Pods we created:

```
$ kubectl get replicasets
No resources found.

$ kubectl get pods
No resources found.
```


In the following example, we are deleting the webserver Deployment which we created in the previous section using the Dashboard

Create a YAML file with Deployment details

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: webserver
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: webserver
        image: nginx:alpine
        ports:
        - containerPort: 80
```


Let us now create the webserver.yaml file with the following content

Create a YAML file with Deployment details

- Using **kubectl**, we will create the Deployment from the YAML file, with the `-f` option to the `kubectl create` command we can pass a YAML file as an object's specification

```
$ kubectl create -f webserver.yaml
deployment "webserver" created
```

- This will also create a ReplicaSet and Pods, as defined.

```
$ kubectl get replicasets
NAME                   DESIRED   CURRENT   READY   AGE
webserver-1529352408   3         3         3       3s

$ kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
webserver-1529352408-5xln6     1/1     Running   0          6s
webserver-1529352408-9st86     1/1     Running   0          6s
webserver-1529352408-tlz4m     1/1     Running   0          6s
```


While creating the deployment, we are using the extensions/v1beta1 API endpoint. With Kubernetes 1.6, a new API group called "apps" was added, and, in the future, the v1 Deployment object would reside there. As of now, both extensions/v1beta1 and apps/v1beta1 are identical

Create a Service and Expose It to the External World with NodePort I

- In the previous section, we took a look at different *ServiceTypes*, with which we can define the access method for a given Service
- For a given Service, with the NodePort *ServiceType*, k8s opens up a static port on all the Worker Nodes
- If we connect to that port from any node, we are forwarded to the respective Service
- Next, let us use the NodePort *ServiceType* and create a Service

Create a Service and Expose It to the External World with NodePort I

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
  labels:
    run: web-service
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: webserver
```

```
$ kubectl create -f webserver-svc.yaml
service "web-service" created
```


Create a webserver-svc.yaml file with the following content

List the Services

```
$ kubectl get service
NAME            CLUSTER       EXTERNAL-IP   PORT(S)         AGE
kubernetes      10.0.0.1      <none>        443/TCP         23d
web-service     10.0.0.133    <nodes>       80:32636/TCP    15s
```

- Our web-service is now created and its ClusterIP is `10.0.0.133`
- In the PORT(S) section we can see a mapping of `80:32636`, which means that we have reserved a static port `32636` on the node
- If we connect to the node on that port, our requests will be forwarded to the ClusterIP on port `80`
- It is not necessary to create the Deployment first, and the Service after. They can be created in any order. A Service will connect Pods based on the Selector

Create a Service and Expose It to the External World with NodePort II

```
$ kubectl describe svc web-service
Name:                   web-service
Namespace:              default
Labels:                 run=web-service
Annotations:            <none>
Selector:               app=webserver
Type:                   NodePort
IP:                     10.0.0.133
Port:                   <unset> 80/TCP
NodePort:               <unset> 32636/TCP
Endpoints:              172.17.0.4:80,172.17.0.5:80,172.17.0.6:80
Session Affinity:       None
Events:                 <none>
```

`web-service` uses `app=webserver` as a Selector, through which it selected our three Pods, which are listed as endpoints. So, whenever we send a request to our Service, it will be served by one of the Pods listed in the `Endpoints` section

To get more details about the Service, we can use the kubectl describe command.

Accessing the Application Using the Exposed NodePort

- Our application is running inside the minikube VM
- To access the application from our workstation, let's first get the IP address of the minikube VM

```
$ minikube ip
192.168.99.100
```

- Now, open the browser and access the application on `192.168.99.100` at port `32636`

Accessing the Application Using the Exposed NodePort

- We can see the *Nginx* welcome page, through which we can access a `webserver` running inside one of the Pods created
- Our requests can be served by any one of the three endpoints selected via our Service

Kubernetes Volume Management

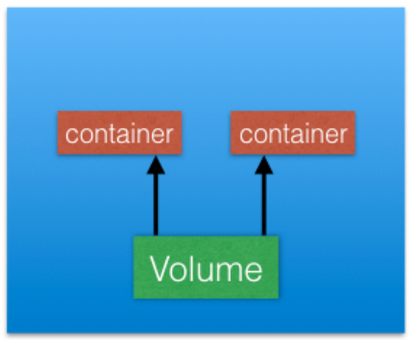
To back a Pod with a persistent storage, Kubernetes uses Volumes. In this section, we will learn about Volumes and their types. We will also discuss about Persistent Volume and Persistent Volume Claim objects, which help us attach storage volumes to Pods.

Volumes

- All data stored inside a container is deleted if the container crashes
  - However, `kubelet` will restart it with a clean state, which means that it will not have any of the old data.
- To overcome this problem, k8s uses Volumes
  - A Volume is essentially a directory backed by a storage medium
  - The storage medium and its content are determined by the Volume Type

Volumes

- In k8s, a Volume is attached to a Pod and shared among the containers of that Pod
- The Volume has the same life span as the Pod, and it outlives the containers of the Pod - this allows data to be preserved across container restarts



Pod

Volume Types

- **emptyDir**
  - An `empty` Volume is created for the Pod as soon as it is scheduled on the Worker Node. The Volume's life is tightly coupled with the Pod. If the Pod dies, the content of `emptyDir` is deleted forever
- **hostPath**
  - With the `hostPath` Volume Type, we can share a directory from the host to the Pod. If the Pod dies, the content of the Volume is still available on the host
- **gcePersistentDisk**
  - With the `gcePersistentDisk` Volume Type, we can mount a Google Compute Engine (GCE) persistent disk into a Pod
- **awsElasticBlockStore**
  - With the `awsElasticBlockStore` Volume Type, we can mount an AWS EBS Volume into a Pod

A directory which is mounted inside a Pod is backed by the underlying Volume Type. A Volume Type decides the properties of the directory, like size, content, etc. Some of the Volume Types are

Volume Types

- **nfs**
  - With nfs, we can mount an NFS share into a Pod.
- **iscsi**
  - With iscsi, we can mount an iSCSI share into a Pod.
- **secret**
  - With the `secret` Volume Type, we can pass sensitive information, such as passwords, to Pods. We will take a look at an example in a later chapter.
- **persistentVolumeClaim**
  - We can attach a Persistent Volume to a Pod using a persistentVolumeClaim

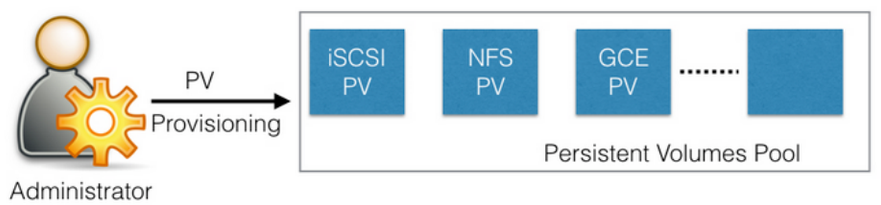
You can learn more details about Volume Types in the Kubernetes Documentation: https://kubernetes.io/docs/concepts/storage/volumes/

Persistent Volumes

- In a typical IT environment, storage is managed by the storage/ system administrators
  - The end user will just get instructions to use the storage, but does not have to worry about the underlying storage management
- In the containerized world, we would like to follow similar rules, but it becomes challenging, given the many Volume Types we have seen earlier
  - K8s resolves this problem with the Persistent Volume subsystem, which provides APIs for users and administrators to manage and consume storage

Persistent Volumes

- To manage the Volume, it uses the PersistentVolume (PV) API resource type, and to consume it, it uses the PersistentVolumeClaim (PVC) API resource type
- A Persistent Volume is a network attached storage in the cluster, which is provisioned by the administrator



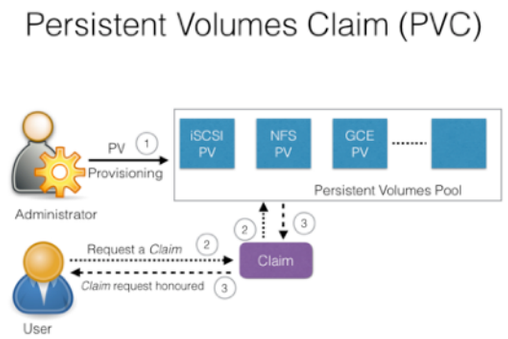
Persistent Volumes

- Persistent Volumes can be provisioned statically by the administrator, or dynamically, based on the StorageClass resource
  - A StorageClass contains pre-defined provisioners and parameters to create a Persistent Volume
- Some of the Volume Types that support managing storage using Persistent Volumes are:
  - GCEPersistentDisk
  - AWSElasticBlockStore
  - AzureFile
  - NFS
  - iSCSI
  - CephFS
  - Cinder

For a complete list, as well as more details, you can check out the Kubernetes Documentation: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistent-volumes

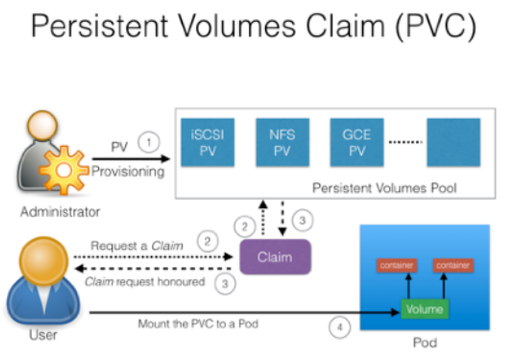
Persistent Volume Claims

- A PersistentVolumeClaim (PVC) is a request for storage by a user
- Users request for Persistent Volume resources based on size, access modes, etc
- Once a suitable Persistent Volume is found, it is bound to a Persistent Volume Claim



Persistent Volume Claims

- After a successful bind, the PersistentVolumeClaim resource can be used in a Pod
- Once a user finishes its work, the attached Persistent Volumes can be released
- The underlying Persistent Volumes can then be reclaimed and recycled for future usage.



To learn more, you can check out the documentation: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims

Deploying a Multi-Tier Application

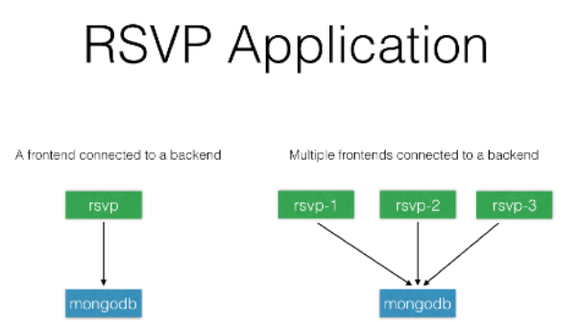
In a typical application, we have different tiers:

- Backend
- Frontend
- Caching, etc.

In this section, we will learn to deploy a multi-tier application with Kubernetes and then scale it.

RSVP Application

- We will be using a sample RSVP application
- Using this application, users can register for an event by providing their username and email id
- Once a user registers, his/her name and email appears in a table
- The application consists of a backend database and a frontend
  - For the backend, we will be using a MongoDB database, and for the frontend, we have a Python Flask-based application



RSVP Application

- The application's code: https://github.com/vinicius3w/rsvpapp
- In the frontend code (`rsvp.py`), we will look for the `MONGODB_HOST` environment variable for the database endpoint, and, if it is set, we will connect to it on port `27017`

```
MONGODB_HOST=os.environ.get('MONGODB_HOST', 'localhost')
client = MongoCLient(MONGODB_HOST, 27017)
```

- After deploying the application with one backend and one frontend, we will scale the frontend to explore the scaling feature of k8s
- Next, we will deploy the MongoDB database - for this, we will need to create a Deployment and a Service for MongoDB

Create the Deployment for MongoDB

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: rsvp-db
spec:
  replicas: 1
  template:
    metadata:
      labels:
        appdb: rsvpdb
    spec:
      containers:
      - name: rsvpd-db
        image: mongo:3.3
        env:
        - name: MONGODB_DATABASE
          value: rsvpdata
        ports:
        - containerPort: 27017
```

```
$ kubectl create -f rsvp-db.yaml
deployment "rsvp-db" created
```


Create an rsvp-db.yaml

Create the Service for MongoDB

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb
  labels:
    app: rsvpdb
spec:
  ports:
  - port: 27017
    protocol: TCP
  selector:
    appdb: rsvpdb
```

```
$ kubectl create -f rsvp-db-service.yaml
service "mongodb" created
```

As we did not specify any ServiceType, `mongodb` will have the default ClusterIP ServiceType
This means that the `mongodb` Service will not be accessible from the external world

To create a mongodb Service for the backend, create an rsvp-db-service.yaml file with the following content

Check Out the Available Deployments and Services

```
$ kubectl get deployments
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
rsvp-db      1         1         1            1           10m
webserver    3         3         3            3           1d

$ kubectl get services
NAME          CLUSTER-IP     EXTERNAL-IP     PORT(S)          AGE
kubernetes    10.0.0.1       <none>          443/TCP          24d
mongodb       10.0.0.32      <none>          27017/TCP        8m
web-service   10.0.0.133     <nodes>         80/32636/TCP     1d
```

Deploy the Python Flask-Based Frontend

- The frontend is created using a Python Flask-based microframework
- Its source code is available here
  - We have created a Docker image called teamcloudyuga/rsvpapp, in which we have imported the application's source code
  - The application's code is executed when a container created from that image runs
  - The Dockerfile to create the `teamcloudyuga/rsvpapp` image is available here

Create the Deployment for the 'rsvp' Frontend

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: rsvp
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: rsvp
    spec:
      containers:
      - name: rsvp-app
        image: teamcloudyuga/rsvpapp
        env:
        - name: MONGODB_HOST
          value: mongodb
        ports:
        - containerPort: 5000
          name: web-port
```

```
$ kubectl create -f rsvp-web.yaml
deployment "rsvp" created
```


To create the rsvp frontend, create an rsvp-web.yaml file

Create the Deployment for the 'rsvp' Frontend

- While creating the Deployment for the frontend, we are passing the name of the MongoDB Service, `mongodb`, as an environment variable, which is expected by our frontend
- Notice that in the `ports` section we mentioned the `containerPort 5000`, and given it the `web-port` name
- We will be using the referenced `web-port` name while creating the Service for the `rsvp` application
- This is useful, as we can change the underlying `containerPort` without making any changes to our Service.

Create the Service for the 'rsvp' Frontend

```
apiVersion: v1
kind: Service
metadata:
  name: rsvp
  labels:
    apps: rsvp
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: web-port
    protocol: TCP
  selector:
    app: rsvp
```

```
$ kubectl create -f rsvp-web-service.yaml
service "rsvp" created
```


To create the rsvp Service for our frontend, create an rsvp-web-service.yaml.
You may notice that we have mentioned the targetPort in the ports section, which will forward all the requests coming on port 80 for the ClusterIP to the referenced web-port port (5000) on the connected Pods. We can describe the Service and verify it.

Check Out the Available Deployments and Services

```
$ kubectl get deployments
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
rsvp         1         1         1            1           40m
rsvp-db      1         1         1            1           1h
webserver    3         3         3            3           1d

$ kubectl get services
NAME          CLUSTER-IP    EXTERNAL-IP   PORT(S)         AGE
kubernetes    10.0.0.1      <none>        443/TCP         25d
mongodb       10.0.0.32     <none>        27017/TCP       1h
rsvp          10.0.0.225    <nodes>       80:31314/TCP    7m
web-service   10.0.0.133    <nodes>       80:32636/TCP    1d
```

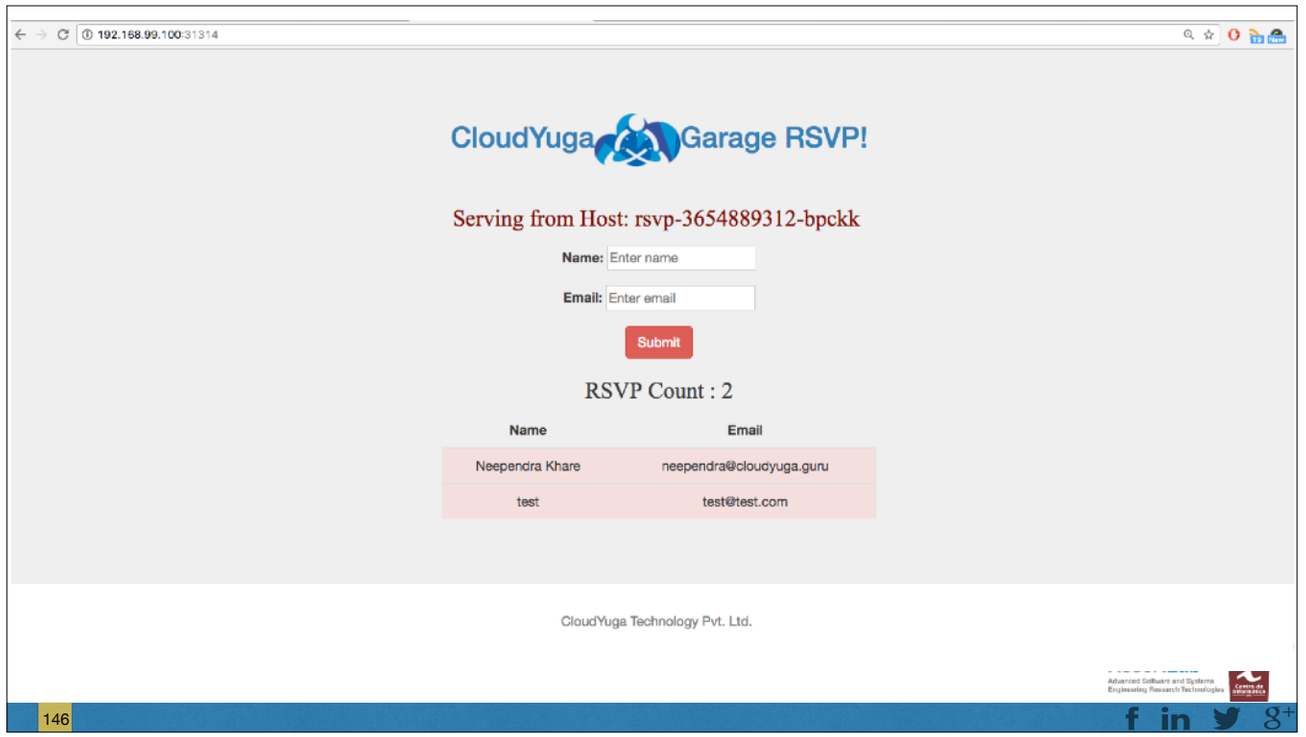

Next, we will check out the Deployments and Services currently available.

Access the RSVP Application from the Workstation

- While deploying the frontend's Service, we have used NodePort as the *ServiceType*, which has configured `port 31314` on the minikube VM to access the application
  - Please note that on your setup, the configured `port` for `NodePort` may be different than ours
  - For this section, please use that respective port wherever we use `port 31314`. Let's now get minikube's IP address:

    ```
    $ minikube ip
    192.168.99.100
    ```
- and open a browser to access the application at `http://192.168.99.100:31314`

Once opened, fill out the required entries using the frontend, which will be saved in the backend database


192.168.99.100:31314
CloudYuga Garage RSVP!
Serving from Host: rsvp-3654889312-bpckk
Name: Enter name
Email: Enter email
Submit
RSVP Count : 2
Name
Email
Neependra Khare
neependra@cloudyuga.guru
test
test@test.com
CloudYuga Technology Pvt. Ltd.

Scale the Frontend

- Currently, we have one replica running for the frontend. To scale it to 4 replicas, we can use the following command:

```
$ kubectl scale --replicas=4 -f /tmp/rsvp-web.yaml
deployment "rsvp" scaled
```

- We can verify if it scaled correctly by looking at the Deployments:

```
$ kubectl get deployments
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
rsvp         4         4         4            4           1h
rsvp-db      1         1         1            1           1h
webserver    3         3         3            3           1d
```

Scale the Frontend

- If we go to the browser and refresh the frontend, the `hostname` printed after the `Serving from Host` text keeps changing: `rsvp-3654889312-b51fn`, `rsvp-3654889312-s7738`, and so on
- This is because the `hostname` is coming from the underlying Pod and, when we refresh the page, we hit different endpoints from our Service

ConfigMaps and Secrets

While deploying an application, we may need to pass such runtime parameters like configuration details, passwords, etc. For example, let's assume we need to deploy ten different applications for our customers, and, for each customer, we just need to change the name of the company in the UI. Then, instead of creating ten different Docker images for each customer, we may just use the template image and pass the customers' names as a runtime parameter. In such cases, we can use the ConfigMap API resource. Similarly, when we want to pass sensitive information, we can use the Secret API resource. In this section, we will explore ConfigMaps and Secrets.

ConfigMaps

- ConfigMaps allow us to decouple the configuration details from the container image
- Using ConfigMaps, we can pass configuration details as key-value pairs, which can be later consumed by Pods, or any other system components, such as controllers
- We can create ConfigMaps in two ways:
  - From literal values
  - From files

Create a ConfigMap from Literal Values and Get Its Details

- A ConfigMap can be created with the `kubectl create` command, and we can get the values using the `kubectl get` command.
- Create the ConfigMap

```
$ kubectl create configmap my-config --
from-literal=key1=value1 --from-
literal=key2=value2
configmap "my-config" created
```

Create a ConfigMap from Literal Values and Get Its Details

```
$ kubectl get configmaps my-config -o yaml
apiVersion: v1
data:
  key1: value1
  key2: value2
kind: ConfigMap
metadata:
  creationTimestamp: 2017-05-31T07:21:55Z
  name: my-config
  namespace: default
  resourceVersion: "241345"
  selfLink: /api/v1/namespaces/default/configmaps/my-config
  uid: d35f0a3d-45d1-11e7-9e62-080027a46057
```


With the -o yaml option, we are requesting the kubectl command to spit the output in the YAML format. As we can see, the object has the ConfigMap kind, and it has the key-value pairs inside the data field. The name of ConfigMap and other details are part of the metadata field

Create a ConfigMap from a Configuration File

- First, we need to create a configuration file. We can have a configuration file with the content like:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: customer1
data:
  TEXT1: Customer1_Company
  TEXT2: Welcomes You
  COMPANY: Customer1 Company Technology Pct. Ltd.
```

- in which we mentioned the `kind`, `metadata`, and `data` fields, which are targeted to connect with the v1 endpoint of the API Server.

Create a ConfigMap from a Configuration File

- If we name the file with the configuration above as `customer1-configmap.yaml`, we can then create the ConfigMap with the following command:

```
$ kubectl create -f customer1-
configmap.yaml
configmap "customer1" created
```

Use ConfigMap Inside Pods

- We can get the values of the given key as **environment variables** inside a Pod
- In the following example, while creating the Deployment, we are assigning values for environment variables from the `customer1` ConfigMap
- We will get the `TEXT1` environment variable set to `Customer1_Company`, `TEXT2` environment variable set to `Welcomes You`, and so on

```
....
containers:
    - name: rsvp-app
      image: teamcloudyuga/rsvpapp
      env:
      - name: MONGODB_HOST
        value: mongodb
      - name: TEXT1
        valueFrom:
          configMapKeyRef:
            name: customer1
            key: TEXT1
      - name: TEXT2
        valueFrom:
          configMapKeyRef:
            name: customer1
            key: TEXT2
      - name: COMPANY
        valueFrom:
          configMapKeyRef:
            name: customer1
            key: COMPANY
....
```

Use ConfigMap Inside Pods

- We can mount a ConfigMap as a Volume inside a Pod
- For each key, we will see a file in the mount path and the content of that file become the respective key's value
- For more details, please study the K8s Documentation

Secrets

- Let's assume that we have a *Wordpress blog* application, in which our `wordpress` frontend connects to the `MySQL` database backend using a password
- While creating the Deployment for `wordpress`, we can put down the `MySQL` password in the Deployment's YAML file, but the password would not be protected
- The password would be available to anyone who has access to the configuration file
- In situations such as the one we just mentioned, the Secret object can help

Secrets

- With Secret, we can share sensitive information like passwords, tokens, or keys in the form of key-value pairs, similar to ConfigMaps; thus, we can control how the information in a Secret is used, reducing the risk for accidental exposures
- In Deployments or other system components, the Secret object is referenced, without exposing its content
- It is important to keep in mind that the Secret data is stored as plain text inside `etcd`
- Administrators must limit the access to the API Server and `etcd`

Create the Secret with the 'kubectl create secret' Command

- To create a Secret, we can use the `kubectl create secret` command:

```
$ kubectl create secret generic my-password
--from-literal=password=mysqlpassword
```

- The above command would create a secret called `my-password`, which has the value of the `password` key set to `mysqlpassword`

'get' and 'describe' the Secret

- Analyzing the `get` and `describe` examples, we can see that they do not reveal the content of the Secret
- The type is listed as `Opaque`

```
$ kubectl get secret my-password
NAME          TYPE      DATA   AGE
my-password   Opaque    1      8m

$ kubectl describe secret my-password
Name:          my-password
Namespace:     default
Labels:        <none>
Annotations:   <none>

Type  Opaque

Data
====
password.txt:  13 bytes
```

Create a Secret Manually

- With Secrets, each object data must be encoded using base64. If we want to have a configuration file for our Secret, we must first get the base64 encoding for our password

```
$ echo mysqlpassword | base64
bXlzcWxwYXNzd29yZAo=
```

- and then use it in the configuration file:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-password
type: Opaque
data:
  password: bXlzcWxwYXNzd29yZAo=
```


We can also create a Secret manually, using the YAML configuration file

ATENTION!!!

- Please note that base64 encoding does not do any encryption, and anyone can easily decode it:

```
$ echo "bXlzcWxwYXNzd29yZAo=" |
base64 --decode
```

- Therefore, make sure you **do not commit** a Secret's configuration file in the source code

Use Secrets Inside Pods

- Using Secrets as Environment Variables
  - As shown in the following example, we can reference a Secret and assign the value of its key as an environment variable (`WORDPRESS_DB_PASSWORD`)

```
.....
    spec:
      containers:
      - image: wordpress:4.7.3-apache
        name: wordpress
        env:
        - name: WORDPRESS_DB_HOST
          value: wordpress-mysql
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-password
              key: password
.....
```

Use Secrets Inside Pods

- Using Secrets as Files from a Pod
  - We can also mount a Secret as a Volume inside a Pod
  - A file would be created for each key mentioned in the Secret, whose content would be the respective value
  - For more details, you can study the k8s Documentation

Ingress

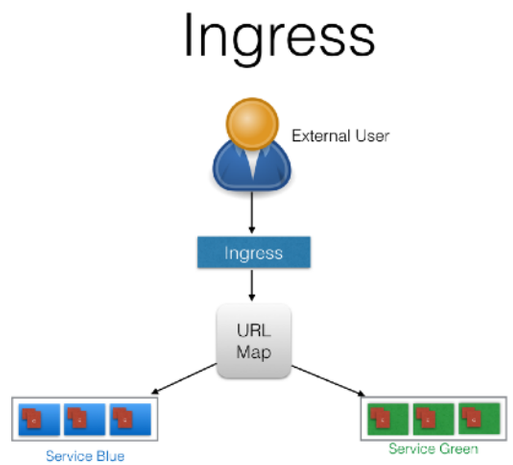
In the Services section, we saw how we can access our deployed containerized application from the external world. Among the ServiceTypes mentioned in that section, NodePort and LoadBalancer are the most often used. For the LoadBalancer ServiceType, we need to have the support from the underlying infrastructure. Even after having the support, we may not want to use it for every Service, as LoadBalancer resources are limited and they can increase costs significantly. Managing the NodePort ServiceType can also be tricky at times, as we need to keep updating our proxy settings and keep track of the assigned ports. In this section, we will explore the Ingress, which is another method we can use to access our applications from the external world.

Ingress

- With Services, routing rules are attached to a given Service
- They exist for as long as the Service exists
  - If we can somehow decouple the routing rules from the application, we can then update our application without worrying about its external access
- This can be done using the Ingress resource
- According to kubernetes.io,
  - "*An Ingress is a collection of rules that allow inbound connections to reach the cluster Services*."

Ingress

- To allow the inbound connection to reach the cluster Services, Ingress configures a Layer 7 HTTP load balancer for Services and provides the following:
  - TLS (Transport Layer Security)
  - Name-based virtual hosting
  - Path-based routing
  - Custom rules



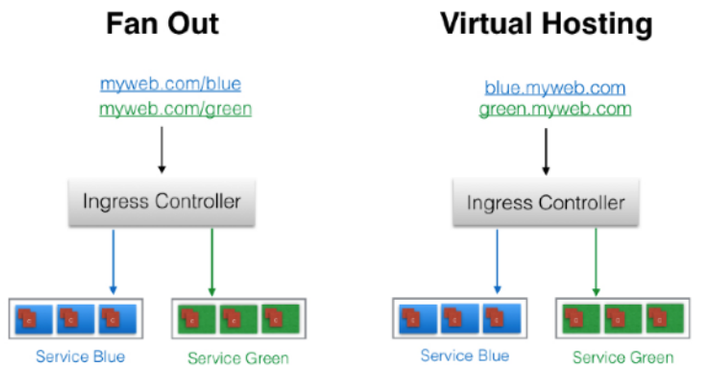
Ingress II

- With Ingress, users don't connect directly to a Service
  - Users reach the Ingress endpoint, and, from there, the request is forwarded to the respective Service
- users requests to both `blue.myweb.com` and `green.myweb.com` would go to the same Ingress endpoint, and, from there, they would be forwarded to `blue-service`, and `green-service`, respectively
- Here, we have seen an example of a Name-Based Virtual Hosting Ingress rule

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: web-ingress
spec:
  rules:
  - host: blue.myweb.com
    http:
      paths:
      - backend:
          serviceName: blue-service
          servicePort: 80
  - host: green.myweb.com
    http:
      paths:
      - backend:
          serviceName: green-service
          servicePort: 80
```

Ingress

- We can also have Fan Out Ingress rules, in which we send requests like `myweb.com/blue` and `myweb.com/green`, which would be forwarded to `blue-service` and `green-service`, respectively
- The Ingress resource does not do any request forwarding by itself
- All of the magic is done using the Ingress Controller



Ingress Controller

- An Ingress Controller is an application which watches the Master Node's API Server for changes in the Ingress resources and updates the Layer 7 load balancer accordingly
- K8s has different Ingress Controllers, and, if needed, we can also build our own
  - GCE L7 Load Balancer and Nginx Ingress Controller are examples of Ingress Controllers
- Start the Ingress Controller with minikube
  - Minikube v0.14.0 and above ships the Nginx Ingress Controller setup as an add-on

```
$ minikube addons enable ingress
```

Deploy an Ingress Resource

- Once the Ingress Controller is deployed, we can create an Ingress resource using the `kubectl create` command
- For example, if we create a `myweb-ingress.yaml` file with the content that we saw on the Ingress II slide, then

```
$ kubectl create -f myweb-ingress.yaml
```

Access Services Using Ingress

- With the Ingress resource we just created, we should now be able to access the `blue-service` or `green-service services` using `blue.myweb.com` and `green.myweb.com` URLs
- As our current setup is on minikube, we will need to update the host configuration file (`/etc/hosts` on Mac and Linux) on our workstation to the minikube's IP for those URLs:
  - `$ cat /etc/hosts`
  - `127.0.0.1  localhost`
  - `::1  localhost`
  - `192.168.99.100  blue.myweb.com green.myweb.com`
- Once this is done, we can now open `blue.myweb.com` and `green.myweb.com` on the browser and access the application

Advanced Topics - Overview

So far, in this course, we have spend most of our time understanding the basic Kubernetes concepts and simple workflows to build a solid foundation. To support enterprise class production workloads, Kubernetes can do auto-scaling, rollbacks, quota management, RBAC, etc. In this section, we will get a high-level overview about such advanced topics, but diving into details would be out of scope for this course.

Annotations

- With Annotations, we can attach arbitrary non-identifying metadata to objects, in a key-value format:

```
"annotations": {
  "key1" : "value1",
  "key2" : "value2"
}
```

- In contrast to Labels, annotations are not used to identify and select objects. Annotations can be used to:
  - Store build/release IDs, PR numbers, git branch, etc.
  - Phone/pager numbers of persons responsible, or directory entries specifying where such information can be found
  - Pointers to logging, monitoring, analytics, audit repositories, debugging tools, etc.
  - Etc.

Annotations

- For example, while creating a Deployment, we can add a description like the one below:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: webserver
  annotations:
    description: Deployment based PoC dates 2nd
June'2017
....
....
```

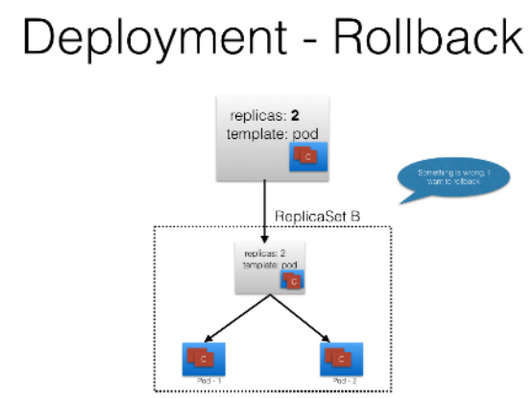
Annotations

- We can look at annotations while describing an object:

```
$ kubectl describe deployment webserver
Name:                   webserver
Namespace:              default
CreationTimestamp:      Sat, 03 Jun 2017 05:10:38 +0530
Labels:                 app=webserver
Annotations:            deployment.kubernetes.io/revision=1
                        description=Deployment based PoC dates 2nd June'2017
...
...
```

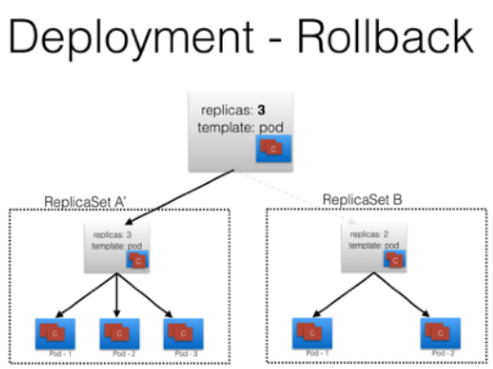
Deployment Features

- Earlier, we have seen how we can use the Deployment object to deploy an application
- We can do more interesting things, like recording a Deployment - if something goes wrong, we can revert to the working state
- The graphic below depicts a situation in which our update fails



Deployment Features

- If we have recorded our Deployment before doing the update, we can revert back to a known working state
- In addition, the Deployment object also provides
  - Autoscaling
  - Proportional scaling
  - Pausing and resuming



Jobs

- A Job creates one or more Pods to perform a given task
- The Job object takes the responsibility of Pod failures
  - It makes sure that the given task is completed successfully
  - Once the task is over, all the Pods are terminated automatically
- Starting with the Kubernetes 1.4 release, we can also perform Jobs at specified times/dates, such as cron jobs

Quota Management

- When there are many users sharing a given k8s cluster, there is always a concern for fair usage
- To address this concern, administrators can use the ResourceQuota object, which provides constraints that limit aggregate resource consumption per Namespace
- We can have the following types of quotas per Namespace:
  - **Compute Resource Quota**: We can limit the total sum of compute resources (CPU, memory, etc.) that can be requested
  - **Storage Resource Quota**: We can limit the total sum of storage resources (persistentvolumeclaims, requests.storage, etc.) that can be requested
  - **Object Count Quota**: We can restrict the number of objects of a given type (pods, ConfigMaps, persistentvolumeclaims, ReplicationControllers, Services, Secrets, etc.)

DaemonSets

- In some cases, like collecting monitoring data from all nodes, or running a storage daemon on all nodes, etc., we need a specific type of Pod running on all nodes at all times
  - A DaemonSet is the object that allows us to do just that
- Whenever a node is added to the cluster, a Pod from a given DaemonSet is created on it
- When the node dies, the respective Pods are garbage collected
- If a DaemonSet is deleted, all Pods it created are deleted as well

StatefulSets

- Before k8s 1.5, the StatefulSet controller was referred to as *PetSet*
- The StatefulSet controller is used for applications which require a unique identity, such as name, network identifications, strict ordering, etc
  - For example, `MySQL cluster`, `etcd cluster`
- The StatefulSet controller provides identity and guaranteed ordering of deployment and scaling to Pods

AssertLab
Advanced Software and Systems Engineering Research Technologies

Role Based Access Control (RBAC)

- Role-based access control (RBAC) is an authorization mechanism for managing permissions around k8s resources
  - It is added as a beta resource in k8s 1.6 release.
- Using the RBAC API, we define a role which contains a set of additive permissions
  - Within a Namespace, a role is defined using the *Role* object
  - For a cluster-wide role, we need to use the *ClusterRole* object
- Once the roles are defined, we can bind them to a user or a set of users using *RoleBinding* and *ClusterRoleBinding*

Kubernetes Federation

- With the K8s Cluster Federation we can manage multiple k8s clusters from a single control plane
  - We can sync resources across the clusters, and have cross cluster discovery
  - This allows us to do Deployments across regions and access them using a global DNS record
- The **Federation** is very useful when we want to build a hybrid solution, in which we can have one cluster running inside our private datacenter and another one on the public cloud
  - We can also assign weights for each cluster in the Federation, to distribute the load as per our choice

Third Party Resources (Objects)

- In most cases, the existing k8s objects are sufficient to fulfill our requirements to deploy an application, but we also have the flexibility to do more
- Using the ThirdPartyResource object type, we can create our own API objects
- In this case, we will need to write a controller, which can listen to their creation/update/deletion
- The controller would then perform operations accordingly

Third Party Resources (Objects)

- For example, the etcd-operator uses the ThirdPartyResource to create objects, and, depending on that, its controller creates/configures/manages `etcd` clusters on top of k8s
- ThirdPartyResource is getting deprecated with Kubernetes 1.7
- A new object type called **Custom Resource Definition (CRD)** was introduced in beta with the K8s 1.7 release, which will help you create new custom resources
- For more details, please checkout out the example from CoreOS

Helm

- To deploy an application, we use different k8s manifests, such as Deployments, Services, Volume Claims, Ingress, etc
- Sometimes, it can be **tiresome** to deploy them one by one
- We can bundle all those manifests after templatizing them into a well-defined format, along with other metadata
- Such a bundle is referred to as Chart
  - These Charts can then be served via repositories, such as those that we have for `rpm` and `deb` packages

Helm

- Helm is a package manager (analogous to `yum` and `apt`) for k8s, which can install/update/delete those Charts in the k8s cluster
- Helm has two components:
  - A client called *helm*, which runs on your user's workstation
  - A server called *tiller*, which runs inside your k8s cluster
- The client *helm* connects to the server *tiller* to manage Charts
  - Charts submitted for k8s are available here

Monitoring and Logging

- In k8s, we have to collect resource usage data by Pods, Services, nodes, etc, to understand the overall resource consumption and to take decisions for scaling a given application
- Two popular k8s monitoring solutions are Heapster and Prometheus
  - Heapster is a cluster-wide aggregator of monitoring and event data, which is natively supported on k8s
  - Prometheus (part of CNCF), can also be used to scrape the resource usage from different k8s components and objects
    - Using its client libraries, we can also instrument the code of our application

Monitoring and Logging

- Another important aspect for troubleshooting and debugging is **Logging**, in which we collect the logs from different components of a given system
- In k8s, we can collect logs from different cluster components, objects, nodes, etc
- The most common way to collect the logs is using Elasticsearch, which uses fluentd with custom configuration as an agent on the nodes. fluentd is an open source data collector, which is also part of CNCF

Kubernetes Community

Just as with any other open source project, the Community plays a vital role in Kubernetes. The Community decides the roadmap of the projects and works towards it. The Community gets engaged in different online and offline forums, like Meetups, Slack, Weekly meetings, etc. In this section, we will explore the Kubernetes Community and see how you can become a part of it, too.

Kubernetes Community

- With more than 35 thousand GitHub stars, k8s is one of the most popular open source projects
  - The Community members not only help with the source code, but they also help with sharing the knowledge
  - The Community engages in both online and offline activities
- Currently, there is a project called K8sPort, which recognizes and rewards Community members for their contributions to k8s
  - This contribution can be in the form of code, attending and speaking at meetups, answering questions on Stack Overflow, etc

Weekly Meetings and Meetup Groups

- **Weekly Meetings**
  - A weekly community meeting happens using video conference tools. You can get a calendar invite from here
- **Meetup Groups**
  - There are many meetup groups around the world, where local community members meet at regular intervals to discuss about k8s and it's ecosystem
- There are some online meetup groups as well, where community members can meet virtually

Slack Channels and Mailing Lists

- **Slack Channels**
  - Community members are very active on the k8s Slack
  - There are different channels based on topics, and anyone can join and participate in the discussions
  - You can discuss with the k8s team on the `#kubernetes-users` channel
- **Mailing Lists**
  - There are k8s users and developers mailing lists, which can be joined by anybody interested

SIGs and Stack Overflow

- **Special Interest Groups**
  - Special Interest Groups (SIGs) focus on specific parts of the k8s project, like scheduling, authorization, Networking, Documentation, etc
  - Each group may have a different workflow, based on its specific requirements
  - A list with all the current SIGs can be found here. Depending on the need, a new SIG can be created
- **Stack Overflow**
  - Besides Slack and mailing lists, Community members can get support from Stack Overflow, as well
  - Stack Overflow is an online environment where you can post questions that you cannot find an answer for
    - The K8s team also monitors the posts tagged **Kubernetes**

CNCF Events

- CNCF organizes numerous international conferences on Kubernetes, as well as other CNCF projects
- Two of the major conferences it organizes are:
  - CloudNativeCon + KubeCon Europe
  - CloudNativeCon + KubeCon North America.

What's Next on Your Kubernetes Journey?

- Now that you have a better understanding of k8s, you can continue your journey by:
  - Participating in activities and discussions organized by the k8s community
  - Attending events organized by the Cloud Native Computing Foundation and The Linux Foundation
  - Expanding your Kubernetes knowledge and skills by enrolling intoners courses (offered by The Linux Foundation for example)
  - Preparing for the upcoming Kubernetes Certified Administrator exam, which will be offered by the Cloud Native Computing Foundation
  - And many other options