kubernetes

Administration

Prerequisites

- ❖ Familiarity with Linux commands
- ❖ Basic knowledge on networking concepts
- ❖ Basic Knowledge on Networking(CIDR blocks, subnet etc.)
- ❖ Target Audience
  - ➢ System administrators
  - ➢ Software developers in a DevOps role
  - ➢ Anyone who wants to learn!!

About You

- Please tell me about yourself:
  - Your Name
  - Your background
  - What is the purpose of this course?
  - Where and how you will be using this knowledge?
  - What do you currently know about Kubernetes?

About me

- ❖ Your Trainer : Deepak Gupta(@hellodk01)
- ❖ Experience : 7+ Years
- ❖ Certifications
  - ➢ Blockchain for Developers
  - ➢ Interfacing with the Raspberry Pi
  - ➢ Big Data, Cloud Computing, & CDN Emerging Technologies

About me

- Industry Roles
  - Devops Lead, MoveinSync
  - Systems Engineer, Myntra Designs
  - Devops Engineer, Knowlarity Communications
  - Software Engineer, Wipro Technologies
- Hobbies : Photography, Travelling, Trekking

Trainings Delivered

- ❖ Cloud Computing : AWS Solutions, Azure DevOps
- ❖ Container Technologies : Docker, Kubernetes
- ❖ Monitoring Tools : Sensu, Zabbix, Nagios, Icinga2
- ❖ SQL Databases : MySQL, PostgreSQL, MariaDB
- ❖ NoSQL Databases : MongoDB, Cassandra, Redis, Gemfire
- ❖ Web Server : Nginx Setup and Configurations
- ❖ Messaging Tools : RabbitMQ, Kafka
- ❖ Configuration Management: Ansible, Chef, Puppet, Saltstack
- ❖ Architecture : Microservices, DevOps, DevSecOps
- ❖ Programming : Java, Python, Golang, haskell

Course Organization

- Hours: 14:00 to 18:00
- Breaks:
  - As and when required!!
- We would be using Centos 7 as our primary OS

Course Organization

- Organize yourself into groups
- Make sure that members of each group sit together
- I hope lab details are already shared with we all

DAY:1

- Introduction
  - Velocity
  - Scaling Service and Teams
  - Abstracting Infrastructure
  - Efficiency
  - Summary
- Creating and Running Containers
  - Container Images
  - Building Application Images with Docker
  - Storing Images in a Remote Registry
  - The Docker Container Runtime
  - Summary

- Deploying a Kubernetes Cluster
  - Installing Kubernetes on a Public Cloud Provider
  - Installing Kubernetes Locally Using minikube
  - The Kubernetes Client
  - Cluster Components
  - Summary

Introduction

- open source orchestrator for deploying containerized applications
- developed by Google
- provides the software necessary to successfully build and deploy
  - reliable
  - scalable distributed systems
- when we say "reliable, scalable distributed systems."
  - More and more services are delivered over the network via APIs
  - APIs are often delivered by a distributed system & must be reliable
- They cannot fail, even if a part of the system crashes or otherwise fails, eg:
  - A live world cup match or a GOT season
  - Aircraft communication systems

Introduction

- Likewise, they must maintain availability even during software rollouts or other maintenance events(cell networks can't go down during upgrade)
- more and more of the world is coming online and using such services
  - services must be highly scalable
  - can grow their capacity to keep up with ever-increasing demand
- many reasons to use containers and container APIs, benefits:
  - Velocity
  - Scaling (of both software and teams)
  - Abstracting our infrastructure
  - Efficiency

Introduction

- velocity – key component in nearly all software development today
- changing nature of software
- boxed software shipped on CDs to web services which change every hour
- difference between us and our competitors is often the speed with which we can develop and deploy new components and features
- this velocity is not defined in terms of simply raw speed
- rather in terms of the number of things we can ship while maintaining a highly available service
- containers and Kubernetes can provide the tools that we need to move quickly, while staying available

Introduction

- The core concepts that enable Velocity are
  - immutability
  - declarative configuration
  - online self-healing systems

Introduction

- immutability
  - artifact created does not change via user modifications
  - rather than a series of incremental updates and changes, an entirely new, complete image is built
  - two ways to upgrade our software:
    - log into a container, update, kill old server,start the new one
      - difficult roll back
    - build new image, kill the existing container, start a new one
      - quick roll back
  - Immutable container images – core of everything that we will build in Kubernetes

Introduction

- declarative configuration
  - declaration of the desired state of the world
  - Everything in Kubernetes is a declarative configuration object
  - configuration object represents the desired state of the system
  - Kubernetes ensures actual state of world matches the desired state
  - an alternative to imperative configuration
  - imperative configuration
    - state of the world is defined by execution of series of instructions
  - imperative commands define actions
  - declarative configurations define state

Introduction

- declarative configuration
  - consider the task of producing three replicas of a piece of software
  - imperative approach
    - configuration would say: “run A, run B, and run C.”
  - declarative approach
    - configuration would be “replicas equals three.”
    - describes the state of the world
    - does not have to be executed to be understood

Introduction

- online self-healing systems
  - Kubernetes initialize our system
  - kubernetes ensures that the current state matches the desired state
  - guard against any failures that might destabilize our system
  - eg:
    - assert a desired state of three replicas to Kubernetes
    - create three replicas & continuously ensures that there are exactly three replicas
    - manually create a fourth replica Kubernetes will destroy one
    - manually destroy a replica, Kubernetes will create one

Introduction

- Scaling our Service and our Teams
  - our product grows, we will need to scale both our software and the teams that develop it
  - Kubernetes achieves scalability by favoring decoupled architectures.
  - Decoupling
    - decoupled architecture each component is separated from other components by defined APIs and service load balancers
    - APIs & load balancers isolate each piece of the system
    - APIs provide a buffer between implementer and consumer
    - load balancers provide a buffer between running instances of each service

Introduction

- Scaling our Service and our Teams
  - Decoupling
    - Decoupling components via load balancers
      - easy to scale the programs that make up our service
    - Decoupling servers via APIs
      - easier to scale the development teams
- decoupling the application container image and machine
- different microservices can collocate on the same machine without interfering with each other
- reduces the overhead and cost of microservice architectures

Introduction

- Kubernetes provides numerous abstractions and APIs that make it easier to build these decoupled microservice architectures
- Pods, or groups of containers, can group together container images developed by different teams into a single deployable unit
- Kubernetes services provide load balancing, naming, and discovery to isolate one microservice from another
- Namespaces provide isolation and access control, so that each microservice can control the degree to which other services interact with it
- Ingress objects provide an easy-to-use frontend that can combine multiple microservices into a single externalized API surface area

- ✓ Kubernetes was built to radically change the way that applications are built and deployed in the cloud
- ✓ Fundamentally, it was designed to give developers more velocity, efficiency, and agility
- ✓ hope we have given we an idea of why we should deploy our applications using Kubernetes
- ✓ Now that we are convinced of that, we will learn how to deploy our application

Kubernetes Terminologies

- Annotation
  - key-value pair used to attach arbitrary non-identifying metadata to objects
- Cluster
  - set of machines/nodes, that run containerized applications
- Container
  - lightweight and portable executable image that contains software and all of its dependencies
- Container Environment Variables
  - name=value pairs that provide useful information into containers running in a Pod

Kubernetes Terminologies

- Controller
  - control loop that watches the shared state of cluster via the apiserver
  - makes changes attempting to move the current state towards the desired state
- CustomResourceDefinition
  - Custom code that defines a resource to add to our Kubernetes API server without building a complete custom server
- DaemonSet
  - Ensures a copy of a Pod is running across a set of nodes in a cluster
- Deployment
  - An API object that manages a replicated application

Kubernetes Terminologies

- Image
  - Stored instance of a container that holds a set of software needed to run an application
- Init
  - ContainerOne or more initialization containers that must run to completion before any app containers run
- Job
  - A finite or batch task that runs to completion
- Kubectl
  - A command line tool for communicating with a Kubernetes API server

Kubernetes Terminologies

- Kubelet
  - agent that runs on each node in the cluster
  - makes sure that containers are running in a pod
- Kubernetes API
  - application serving Kubernetes functionality via RESTful interface
- Label
  - Tags objects with identifying attributes that are meaningful and relevant to users
- Minikube
  - tool for running Kubernetes locally

Kubernetes Terminologies

- Name
  - client-provided string that refers to an object in a resource UR
- Namespace
  - abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster
- Node
  - node is a worker machine in Kubernetes
- Pod
  - smallest & simplest Kubernetes object
  - represents a set of running containers on our cluster

Kubernetes Terminologies

- ❖ Pod Security Policy
  - ➢ Enables fine-grained authorization of Pod creation and updates
- ❖ RBAC (Role-Based Access Control)
  - ➢ Manages authorization decisions, allowing admins to dynamically configure access policies through the Kubernetes API
- ❖ ReplicaSet
  - ➢ ReplicaSet is the next-generation Replication Controller
- ❖ Resource Quotas
  - ➢ constraints to limit aggregate resource consumption per Namespace
- ❖ Selector
  - ➢ Allows users to filter a list of resources based on labels

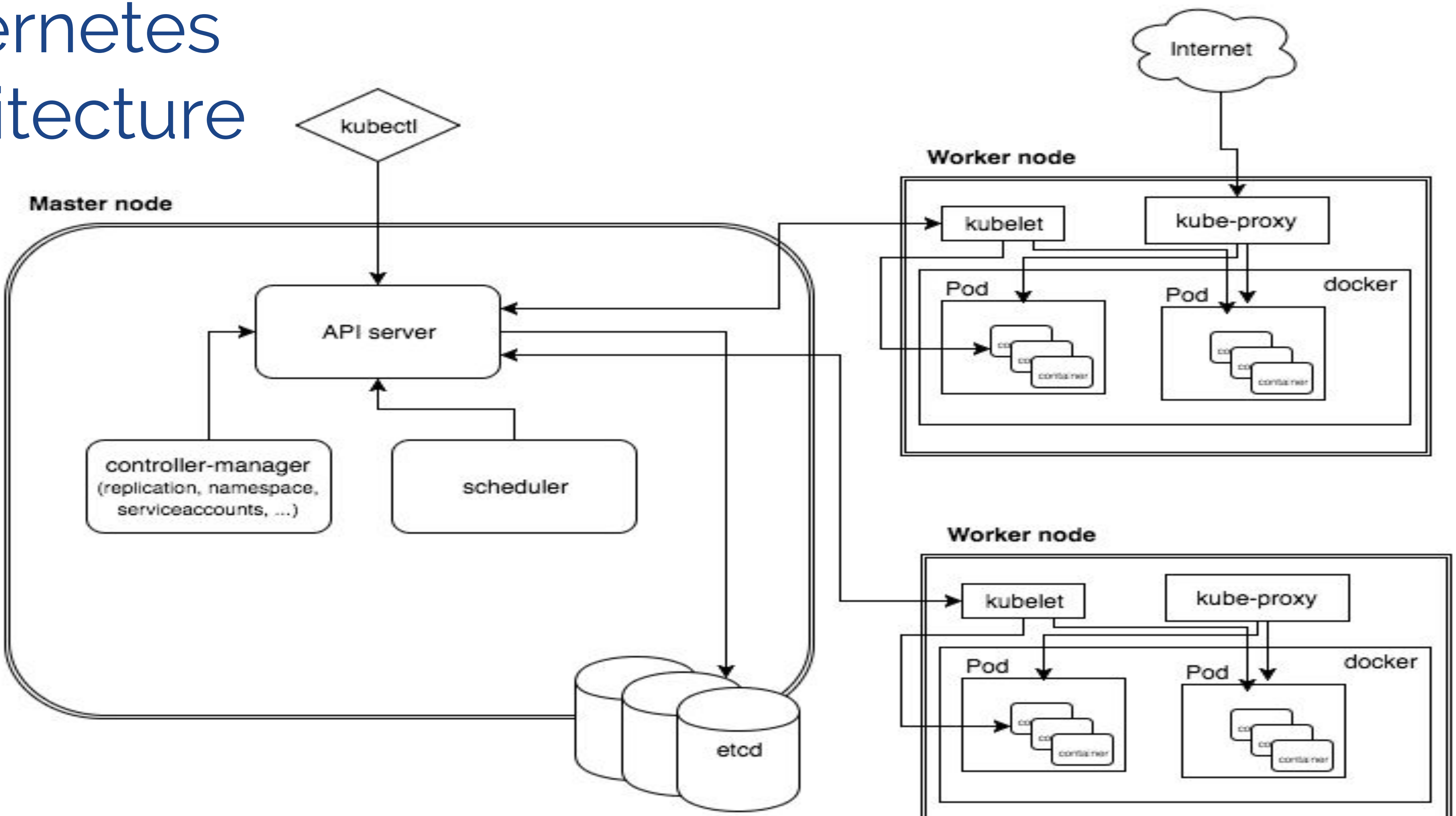
Kubernetes Terminologies

- Service
  - An API object that describes how to access applications, such as a set of Pods , and can describe ports and load-balancers
- Service Account
  - Provides an identity for processes that run in a Pod
- StatefulSet
  - Manages the deployment and scaling of a set of Pods , and provides guarantees about the ordering and uniqueness of these Pods
- UID
  - A Kubernetes systems-generated string to uniquely identify objects

Kubernetes Terminologies

- Volume
  - directory containing data, accessible to the containers in a pod
- kube-apiserver
  - Component on the master that exposes the Kubernetes API. It is the front-end for the Kubernetes control plane
- kube-controller-manager
  - Component on the master that runs controllers
- kube-proxy
  - kube-proxy is a network proxy that runs on each node in the cluster

Kubernetes Architecture



Creating and Running Containers

- Kubernetes – platform for creating, deploying, managing applications
- consider how to build the application container images that make up the pieces of our distributed system
- Applications comprise of a language runtime, libraries, our source code
- also application relies on external libraries such as libc and libssl
- external libraries are generally shipped as shared components in the OS
- Problems occur when application is developed on a programmer's laptop
- app has a dependency on a shared library that isn't available when the program is rolled out to the production OS
- problems can occur when developers forget to include dependent asset files inside a package that they deploy to production

Creating and Running Containers

- A program can only execute successfully if it can be reliably deployed onto the machine where it should run
- we are going to work with a simple example application that we built
- git clone https://github.com/practice02/kuard.git
- cd kuard
- Dockerfile
  - FROM alpine

    MAINTAINER Deepak Gupta <hello.dk@outlook.com>

    COPY bin/kuard /kuard

    ENTRYPOINT ["/kuard"]

Creating and Running Containers

- Ensure we have a kuard executable file inside the bin folder by running make inside the kuard directory
  - make
  - ls -ltr bin #verify if the binary file has been created
  - docker build -t hellodk/kuard-amd64:1 .
  - docker tag kuard-amd64:1 hellodk/kuard-amd64:1
  - docker push hellodk/kuard-amd64:1
  - docker run -d --name kuard --publish 8080:8080 hellodk/kuard-amd64:1
  - curl http://localhost:8080

Creating and Running Containers

- A program can only execute successfully if it can be reliably deployed onto the machine where it should run
- we are going to work with a simple example application that we built

Limiting Resource Usage

- limit the amount of resources used by applications by exposing the underlying cgroup technology provided by the Linux kernel
- key benefits to running applications within a container is the ability to restrict resource utilization
- allows multiple applications to coexist on the same hardware & fair usage

Limiting Resource Usage

- LIMITING MEMORY RESOURCES
  - limit kuard to 200 MB of memory and 1 GB of swap space, use the --memory and --memory-swap flags with the docker run command
  - Stop and remove the current kuard container:
    - docker stop kuard
    - docker rm kuard
  - start another kuard container using the appropriate flags to limit memory usage:
    - docker run -d --name kuard --publish 8080:8080 --memory 200m --memory-swap 1G hellodk/kuard-amd64:1

Limiting Resource Usage

- LIMITING CPU RESOURCES
  - Another critical resource on a machine is the CPU. Restrict CPU utilization using the --cpu-shares flag with the docker run command:
    - docker run -d --name kuard --publish 8080:8080 --memory 200m --memory-swap 1G --cpu-shares 1024 hellodk/kuard-amd64:1
- Cleanup – delete it with the docker rmi command(via their tag name or ID)
  - docker rmi <tag-name>
  - docker rmi <image-id>
- image ID can be shortened as long as it remains unique
- Generally only three or four characters of the ID are necessary

Limiting Resource Usage

- unless we explicitly delete an image it will live on our system forever
- even if we build a new image with an identical name
- Building new image simply moves the tag to the new image
- it doesn't delete or replace the old image
- as we iterate while we are creating a new image
- end up taking up unnecessary space on our computer
- use the docker images command then delete tags we are no longer using

- ✓ Application containers provide a clean abstraction for applications
- ✓ when packaged in the Docker image format, applications become easy to build, deploy, and distribute
- ✓ Containers also provide isolation between applications running on the same machine, which helps avoid dependency conflicts

Deploying a Kubernetes Cluster

- Now that we have successfully built an application container
- how to deploy it into complete reliable, scalable distributed system?
- there are several cloud-based Kubernetes services that make it easy to create a cluster with a few command-line instructions
- local development can be more attractive – minikube tool
- minikube only creates a single-node cluster
- Installing Kubernetes on a Public Cloud Provider

Deploying a Kubernetes Cluster

- Google Container Service
  - hosted Kubernetes-as-a-Service - Google Container Engine (GKE)
  - Once we have gcloud installed, first set a default zone:
    - gcloud config set compute/zone us-west1-a
  - Then we can create a cluster:
  - gcloud container clusters create kuar-cluster
  - When the cluster is ready we can get credentials for the cluster using:
    - gcloud auth application-default login
  - run into trouble?
    - complete instructions for creating a GKE cluster can be found in the Google Cloud Platform documentation

Deploying a Kubernetes Cluster

- Installing Kubernetes with Azure Container Service
  - hosted Kubernetes-as-a-Service as part of the Azure Container Service
  - we can activate the shell by clicking the shell icon:
    - azure-cloud-console.png
  - install the az command-line interface (CLI) on our local machine
  - Once we have the shell up and working, we can run:
    - az group create --name=kuar --location=westus
  - Once the resource group is created, we can create a cluster using:
    - az acs create --orchestrator-type=kubernetes --resource-group=kuar --name=kuar-cluster

Deploying a Kubernetes Cluster

- Once the cluster is created, we can get credentials for the cluster with:
  - az acs kubernetes get-credentials --resource-group=kuar --name=kuar-cluster
- If we don't already have the kubectl tool installed, we can install it using:
  - az acs kubernetes install-cli
- Complete instructions for installing Kubernetes on Azure can be found in the Azure documentation

Deploying a Kubernetes Cluster

- To start with our hands-on, we will use minikube
  - single-node cluster using minikube
  - minikube is a good simulation of a Kubernetes cluster
  - intended for local development, learning, and experimentation
  - it only runs in a VM on a single node
  - it doesn't provide the reliability of a distributed Kubernetes cluster

Deploying a Kubernetes Cluster

- NOTE
  - hypervisor installed on our machine to use minikube
  - For Linux and macOS, this is generally virtualbox
  - Make sure we install the hypervisor before using minikube
- Install
  - brew cask install minikube kubernetes-cli
  - linux
    - curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
    - sudo install minikube-linux-amd64 /usr/local/bin/minikube
    - echo "/usr/local/bin" >> /etc/bashrc # then logout and re-login

Deploying a Kubernetes Cluster

- create a local VM, provision Kubernetes, and create a local kubectl configuration that points to that cluster
  - minikube start
- stop the VM with:
  - minikube stop
- delete the cluster
  - minikube delete
- kubectl version
- minikube version

Deploying a Kubernetes Cluster

- minikube start
- kubectl run hello-minikube --image=k8s.gcr.io/echoserver:1.10 --port=8080
- kubectl expose deployment hello-minikube --type=NodePort
- kubectl get pod
- kubectl get pod
- curl $(minikube service hello-minikube --url)
- kubectl delete services hello-minikube
- kubectl delete deployment hello-minikube
- minikube stop

Deploying a Kubernetes Cluster

- The Kubernetes Client
  - official Kubernetes client is kubectl
  - kubectl – command-line tool for interacting with the Kubernetes API
  - kubectl can be used to manage most Kubernetes objects such as pods, ReplicaSets, and services
  - can also be used to explore and verify the overall health of the cluster

Deploying a Kubernetes Cluster

- Checking Cluster Status
  - first thing we can do is check the version of the cluster
    - kubectl version
  - displays two different versions:
    - version of the local kubectl tool
    - version of the Kubernetes API server
  - Don't worry if these versions are different
  - The Kubernetes tools are backward- and forward-compatible with different versions of the Kubernetes API, so long as we stay within two minor versions of the tools and the cluster and don't try to use newer features on an older cluster

Deploying a Kubernetes Cluster

- diagnostic for the cluster – verify that our cluster is generally healthy:
  - kubectl get componentstatuses
  - output should look like this:

NAME	STATUS	MESSAGE	ERROR
scheduler	Healthy	ok	
controller-manager	Healthy	ok	
etcd-0	Healthy	{"health": "true"}	

Deploying a Kubernetes Cluster

- controller-manager – responsible for running various controllers that regulate behavior in the cluster
  - eg: ensuring that all of the replicas of a service are available & healthy
- scheduler is responsible for placing different pods onto different nodes in the cluster
- etcd server is the storage for the cluster where all of the API objects are stored

Deploying a Kubernetes Cluster

- Listing Kubernetes Worker Nodes
- kubectl get nodes

NAME	STATUS	AGE	VERSION
kubernetes	Ready,master	45d	v1.7.6
node-1	Ready	45d	v1.7.6
node-2	Ready	45d	v1.7.6
node-3	Ready	45d	v1.7.6

- In Kubernetes nodes are separated into master nodes
  - contain containers like the API server, scheduler, etc., which manage the cluster, and worker nodes where our containers will run
- Kubernetes won't generally schedule work onto master nodes to ensure that user workloads don't harm the overall operation of the cluster

Deploying a Kubernetes Cluster

- use the kubectl describe command to get more information about a specific node such as node-1:
  - kubectl describe nodes node-1
- First, we see basic information about the node:

  Name: node-1

  Role:

  Labels: beta.kubernetes.io/arch=arm

  beta.kubernetes.io/os=linux

  kubernetes.io/hostname=node-1
- we can see that this node is running the Linux OS & is running on an ARM processor

Deploying a Kubernetes Cluster

- Next, we see information about the operation of node-1 itself:

```
Conditions:
 Type           Status LastHeartbeatTime  Reason                      Message
 ----           ------ -----------------  ------                      -------
 OutOfDisk      False  Sun, 05 Feb 2017... KubeletHasSufficientDisk   kubelet...
 MemoryPressure False  Sun, 05 Feb 2017... KubeletHasSufficientMemory
kubelet...
 DiskPressure   False  Sun, 05 Feb 2017... KubeletHasNoDiskPressure
kubelet...
 Ready          True   Sun, 05 Feb 2017... KubeletReady               kubelet...
```

Deploying a Kubernetes Cluster

- These statuses show that the node has sufficient disk and memory space, and it is reporting that it is healthy to the Kubernetes master
- Next, there is information about the capacity of the machine:

```
Capacity:
 alpha.kubernetes.io/nvidia-gpu:     0
 cpu:                    4
 memory:                     882636Ki
 pods:                    110
Allocatable:
 alpha.kubernetes.io/nvidia-gpu:     0
 cpu:                    4
 memory:                     882636Ki
 pods:                    110
```

Deploying a Kubernetes Cluster

- Then, there is information about the software on the node, version of Docker running, the versions of Kubernetes & the Linux kernel etc.

  System Info:

Machine ID:	9989a26f06984d6dbadc01770f018e3b
System UUID:	9989a26f06984d6dbadc01770f018e3b
Boot ID:	98339c67-7924-446c-92aa-c1bfe5d213e6
Kernel Version:	4.4.39-hypriotos-v7+
OS Image:	Raspbian GNU/Linux 8 (jessie)
Operating System:	linux
Architecture:	arm
Container Runtime Version:	docker://1.12.6
Kubelet Version:	v1.5.2
Kube-Proxy Version:	v1.5.2

  PodCIDR: 10.244.2.0/24

  ExternalID: node-1

Deploying a Kubernetes Cluster

- Finally, there is information about the pods that are currently running on this node:

```
Non-terminated Pods:        (3 in total)
  Namespace   Name     CPU Requests CPU Limits Memory Requests Memory Limits
  ---------   ----     ------------ ---------- --------------- -------------
  kube-system kube-dns...  260m (6%)    0 (0%)      140Mi (16%)     220Mi (25%)
  kube-system kube-fla...  0 (0%)       0 (0%)      0 (0%)          0 (0%)
  kube-system kube-pro...  0 (0%)       0 (0%)      0 (0%)          0 (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.
  CPU Requests  CPU Limits    Memory Requests Memory Limits
  ------------  ----------    --------------- -------------
  260m (6%)     0 (0%)        140Mi (16%)     220Mi (25%)
```

Deploying a Kubernetes Cluster

- No events
  - From this output we can see the pods on the node (e.g., the kube-dns pod that supplies DNS services for the cluster), the CPU and memory that each pod is requesting from the node, as well as the total resources requested
- Kubernetes tracks both the request and upper limit for resources for each pod that runs on a machine
- resources requested by a pod are guaranteed to be present on the node
- a pod's limit – maximum amount of a given resource a pod can consume
- A pod's limit can be higher than its request, in which case the extra resources are supplied on a best-effort basis

Deploying a Kubernetes Cluster

- Cluster Components
- interesting aspects of Kubernetes is that many of the components that make up the Kubernetes cluster are actually deployed using Kubernetes itself
- All of these components run in the kube-system namespace
- Kubernetes Proxy
  - routing network traffic to load-balanced services in the cluster
  - the proxy must be present on every node in the cluster
  - Kubernetes has an API object named DaemonSet, used in many clusters to accomplish this

Deploying a Kubernetes Cluster

- Kubernetes Proxy
  - If our cluster runs the Kubernetes proxy with a DaemonSet, we can see the proxies by running:
    - kubectl get daemonSets --namespace=kube-system kube-proxy

NAME	DESIRED	CURRENT	READY	NODE-SELECTOR	AGE
kube-proxy	4	4	4	<none>	45d

Deploying a Kubernetes Cluster

- Kubernetes DNS
  - Kubernetes also runs a DNS server
  - provides naming and discovery for the services defined in the cluster
  - This DNS server also runs as a replicated service on the cluster
  - Depending on the size of our cluster, we may see one or more DNS servers running in our cluster
  - DNS service is run as a Kubernetes deployment, which manages these replicas:
    - kubectl get deployments --namespace=kube-system kube-dns

```
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube-dns  1         1         1            1           45d
```

Deploying a Kubernetes Cluster

- Kubernetes DNS
  - There is also a Kubernetes service that performs load-balancing for the DNS server:
    - kubectl get services --namespace=kube-system kube-dns

      NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE

      kube-dns 10.96.0.10 <none> 53/UDP,53/TCP 45d
  - This shows that the DNS service for the cluster has the address 10.96.0.10
  - If we log into a container in the cluster, we'll see that this has been populated into the /etc/resolv.conf file for the container

Deploying a Kubernetes Cluster

- Kubernetes UI
  - final Kubernetes component is a GUI
  - UI is run as a single replica, still managed by a Kubernetes deployment for reliability and upgrades
  - we can see this UI server using:
    - kubectl get deployments --namespace=kube-system kubernetes-dashboard

      NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

      kubernetes-dashboard 1 1 1 1 45d

Deploying a Kubernetes Cluster

- Kubernetes UI
  - The dashboard also has a service that performs load balancing for the dashboard:
  - kubectl get services --namespace=kube-system kubernetes-dashboard

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes-dashboard	10.99.104.174	<nodes>	80:32551/TCP	45d

  - We can use the kubectl proxy to access this UI
    - kubectl proxy

      This starts up a server running on localhost:8001
  - use this interface to explore our cluster, create new containers

✓ Hopefully at this point we have a Kubernetes cluster

✓ we've used a few commands to explore the cluster we have created

✓ Next, we'll spend some more time exploring the command-line interface to that Kubernetes cluster using kubectl tool


Break

DAY: 2

Day 2

- ❖ Common kubectl Commands
  - ➢ Namespaces
  - ➢ Contexts
  - ➢ Viewing Kubernetes API Objects
  - ➢ Creating, Updating, and Destroying Kubernetes Objects
  - ➢ Labelling and Annotating Objects
  - ➢ Debugging Commands
  - ➢ Summary

Day 2

- Pods
  - Pods in Kubernetes
  - Thinking with Pods
  - The Pod Manifest
  - Running Pods
  - Accessing Pod
  - Health Checks
  - Resource Management
  - Persisting Data with Volumes
  - Putting It All Together
  - Summary

Day 2

- ❖ Labels and Annotations
  - ➢ Labels
  - ➢ Annotations
  - ➢ Summary

Common kubectl Commands

- kubectl command-line utility is a powerful tool
- we will use it to create objects and interact with the Kubernetes API
- we will cover basic kubectl commands that apply to all Kubernetes objects
- Namespaces
  - Kubernetes uses namespaces to organize objects in the cluster
  - think of each namespace as a folder that holds a set of objects
  - kubectl command-line tool interacts with the default namespace
  - want to use a different namespace? pass kubectl the --namespace flag
    - kubectl --namespace=mystuff
    - above command references objects in the mystuff namespace

Common kubectl Commands

- Contexts
  - want to change the default namespace more permanently?
  - use a context
  - This gets recorded in a kubectl configuration file, usually located at $HOME/.kube/config
  - This configuration file also stores how to both find and authenticate to our cluster
  - Eg: we can create a context with a different default namespace for our kubectl commands using:
    - kubectl config set-context my-context --namespace=mystuff

Common kubectl Commands

- Contexts
  - This creates a new context, but it doesn't actually start using it yet
  - To use this newly created context, we can run:
    - kubectl config use-context my-context
  - Contexts can also be used to manage different clusters or different users for authenticating to those clusters using the --users or --clusters flags with the set-context command

Common kubectl Commands

- Viewing Kubernetes API Objects
  - Everything contained in Kubernetes is represented by RESTful resource
  - Each Kubernetes object exists at a unique HTTP path
  - eg:
    - https://our-k8s.com/api/v1/namespaces/default/pods/my-pod leads to the representation of a pod in the default namespace named my-pod
  - kubectl command makes HTTP requests to these URLs to access the Kubernetes objects that reside at these paths

Common kubectl Commands

- Viewing Kubernetes API Objects
  - most basic command for viewing Kubernetes objects via kubectl is get
  - run kubectl get <resource-name> we will get a listing of all resources in the current namespace
  - If we want to get a specific resource, we can use kubectl get <resource-name> <object-name>
  - By default, kubectl uses a human-readable printer for viewing the responses from the API server
  - printer removes many of the details of the objects to fit each object on one terminal line

Common kubectl Commands

- Viewing Kubernetes API Objects
  - add the -o wide flag, which gives more details, on a longer line
  - want to view the complete object?
  - view the objects as raw JSON or YAML using the -o json or -o yaml flags, respectively
  - common option for manipulating the output of kubectl is to remove the headers, which is often useful when combining kubectl with Unix pipes (e.g., kubectl ... | awk ...)
  - specify the --no-headers flag, kubectl will skip the headers at the top of the human-readable table

Common kubectl Commands

- Viewing Kubernetes API Objects
  - Another common task is extracting specific fields from the object
  - kubectl uses the JSONPath query language to select fields in object
  - an example, this command will extract and print the IP address of the pod:
    - kubectl get pods my-pod -o jsonpath --template={.status.podIP}
  - interested in more detailed information about a particular object?
    - kubectl describe <resource-name> <obj-name>

Common kubectl Commands

- ❖ Viewing Kubernetes API Objects
  - ➢ Creating, Updating, and Destroying Kubernetes Objects
  - ➢ Objects in the Kubernetes API are represented as JSON or YAML files
  - ➢ These files are either returned by the server in response to a query or posted to the server as part of an API request
  - ➢ we can use these YAML or JSON files to create, update, or delete objects on the Kubernetes server

Common kubectl Commands

- Viewing Kubernetes API Objects
  - Let's assume that we have a simple object stored in obj.yaml
  - use kubectl to create/update this object in Kubernetes by running:
    - kubectl apply -f obj.yaml
  - making interactive edits, instead of editing a local file?
  - use the edit command, which downloads the latest object state, & then launch an editor that contains the definition:
    - kubectl edit <resource-name> <obj-name>
  - save file, it will automatically be uploaded back to the Kubernetes cluster

Common kubectl Commands

- Viewing Kubernetes API Objects
  - delete an object, we can simply run:
    - kubectl delete -f obj.yaml
  - delete an object using the resource type and name:
    - kubectl delete <resource-name> <obj-name>

Common kubectl Commands

- Labeling and Annotating Objects
  - Labels and annotations are tags for our objects
  - update the labels and annotations on any Kubernetes object
  - Eg: to add the color=red label to a pod named bar, we can run:
    - kubectl label pods bar color=red
  - By default, label and annotate will not let us overwrite an existing label
  - To do this, we need to add the --overwrite flag
  - remove a label, we can use the -<label-name> syntax:
    - kubectl label pods bar color-
  - This will remove the color label from the pod named bar

Common kubectl Commands

- Labeling and Annotating Objects
  - Labels and annotations are tags for our objects
  - update the labels and annotations on any Kubernetes object
  - Eg: to add the color=red label to a pod named bar, we can run:
    - kubectl label pods bar color=red
  - By default, label and annotate will not let us overwrite an existing label
  - To do this, we need to add the --overwrite flag
  - remove a label, we can use the -<label-name> syntax:
    - kubectl label pods bar color-
  - This will remove the color label from the pod named bar

Common kubectl Commands

- Debugging Commands
  - number of commands available for debugging our containers
  - we can use the following to see the logs for a running container:
    - kubectl logs <pod-name>
  - multiple containers in our pod?
    - choose the container to view using the -c flag
  - By default, kubectl logs lists the current logs and exits
  - If we instead want to continuously stream the logs back to the terminal without exiting, we can add the -f (follow) command-line flag

Common kubectl Commands

❖ Debugging Commands

➢ use the exec command to execute a command in a running container:

- kubectl exec -it <pod-name> -- bash

➢ This will provide we with an interactive shell inside the running container so that we can perform more debugging

➢ Finally, we can copy files to and from a container using the cp command:

- kubectl cp <pod-name>:/path/to/remote/file /path/to/local/file

➢ This will copy a file from a running container to our local machine

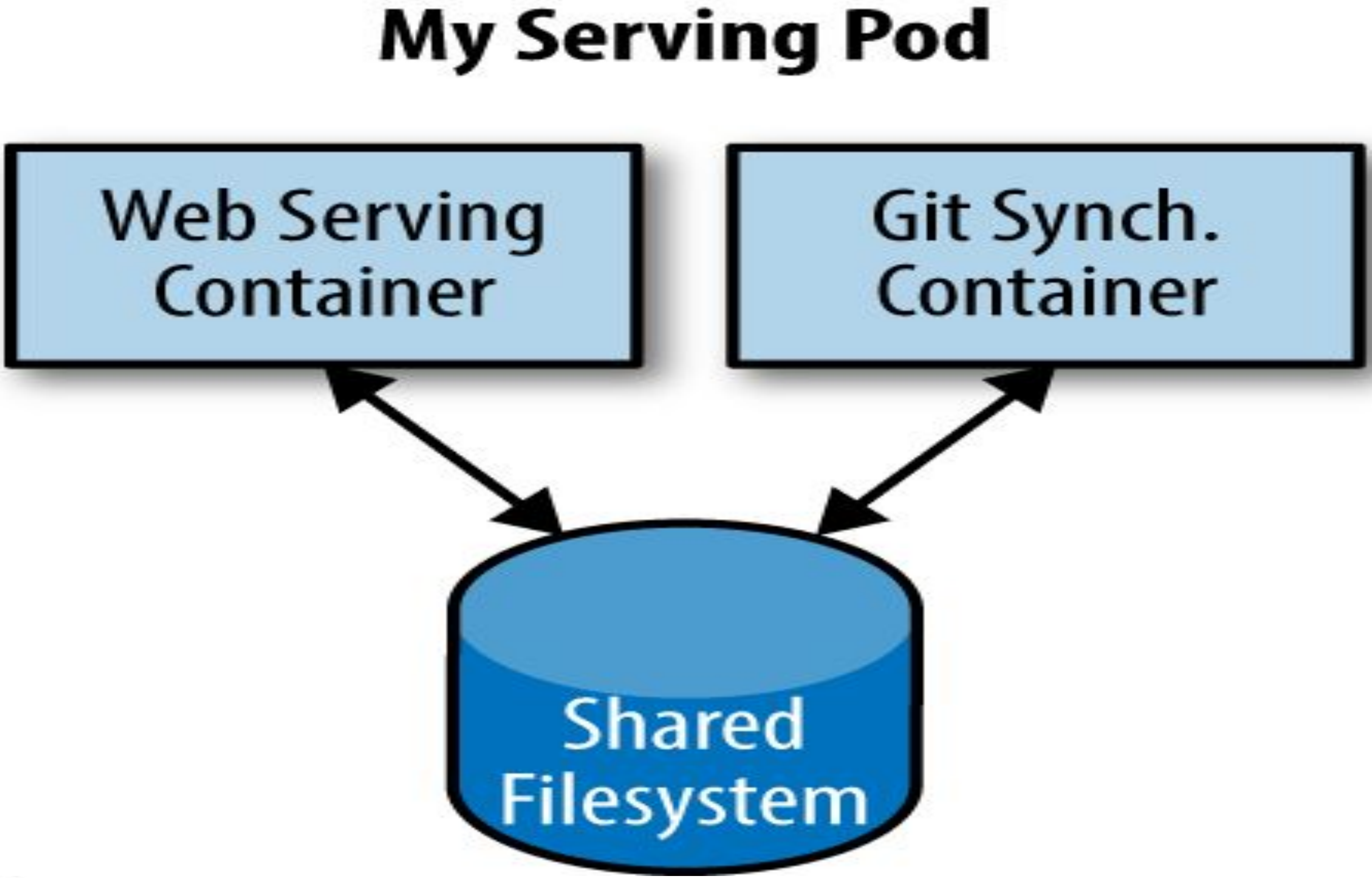
➢ we can also specify directories, or reverse the syntax to copy a file from our local machine back out into the container

- ✓ kubectl is a powerful tool for managing our applications in our Kubernetes cluster
- ✓ illustrated many of the common uses for the tool, but kubectl has a great deal of built-in help available
- ✓ we can start viewing this help with:
  - ➢ kubectl help
  - ➢ or:
  - ➢ kubectl help command-name

Pods

- in real-world deployments of containerized applications we will often want to colocate multiple applications into a single atomic unit, scheduled onto a single machine
- A canonical example of such a deployment is illustrated in next slide
- consists of a container serving web requests and a container synchronizing the filesystem with a remote Git repository

Pods


My Serving Pod
Web Serving Container
Git Synch. Container
Shared Filesystem

Pods

- A Pod represents a collection of application containers and volumes running in the same execution environment
- Pods, not containers, are the smallest deployable artifact in a Kubernetes cluster
- means all of the containers in a Pod always land on the same machine
- Each container within a Pod runs in its own cgroup
- Applications running in the same Pod share the same IP address and port space (network namespace), have the same hostname (UTS namespace), and can communicate using native interprocess communication channels over System V IPC or POSIX message queues (IPC namespace)

Pods

- applications in different Pods are isolated from each other
- they have different IP addresses, different hostnames, and more
- What should I put in a Pod?
  - A WordPress container & a MySQL database container in the same Pod
    - an antipattern for Pod construction
  - two reasons for this
    - Wordpress and its database are not truly symbiotic
    - If the WordPress container and the database container land on different machines, they still can work together quite effectively, since they communicate over a network connection

Pods

- the right question to ask when designing Pods is
  - Will these containers work correctly if they land on different machines?
  - If the answer is “no,” a Pod is the correct grouping for the containers
  - If the answer is “yes,” multiple Pods is probably the correct solution

Pods

- The Pod Manifest
  - Pods described in a Pod manifest
  - a text-file representation of the Kubernetes API object
  - Kubernetes API server accepts and processes Pod manifests before storing them in persistent storage (etcd)
  - scheduler also uses the Kubernetes API to find Pods that haven't been scheduled to a node
  - scheduler then places the Pods onto nodes depending on the resources and other constraints expressed in the Pod manifests
  - Multiple Pods can be placed on the same machine as long as there are sufficient resources

Pods

- scheduling multiple replicas of the same application onto the same machine is worse for reliability, since the machine is a single failure domain
- Once scheduled to a node, Pods don’t move and must be explicitly destroyed and rescheduled
- Multiple instances of a Pod can be deployed by repeating the workflow described here

Pods

- Creating a Pod
  - simplest way to create a Pod is via the imperative kubectl run command
  - to run our same kuard server, use:
    - kubectl run kuard --image=hellodk/kuard-amd64:1
  - see the status of this Pod by running:
    - kubectl get pods
  - delete this Pod by running:
    - kubectl delete deployments/kuard

Pods

- Creating a Pod Manifest
  - written using YAML or JSON
  - YAML is preferred
    - slightly more human-editable
    - has the ability to add comments
  - Pod manifests include a couple of key fields and attributes:
    - mainly a metadata section for describing the Pod and its labels
    - a spec section for describing volumes
    - a list of containers that will run in the Pod

Pods

- deployed kuard using the following Docker command:
  - docker run -d --name kuard --publish 8080:8080 hellodk/kuard-amd64:1
- similar result can be achieved by instead writing to a file named kuard-pod.yaml and then using kubectl commands to load that manifest to Kubernetes

Pods

- kuard-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: hellodk/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Pods

- Running Pods
  - kubectl apply -f kuard-pod.yaml
- Pod manifest will be submitted to the Kubernetes API server
- Kubernetes system will then schedule that Pod to run on a healthy node in the cluster, where it will be monitored by the kubelet daemon process
- Listing Pods
  - kubectl get pods

```
NAME     READY   STATUS    RESTARTS   AGE
kuard    1/1     Running   0          44s
```

- Adding -o wide to any kubectl command will print out more information
- Adding -o json or -o yaml will print out the complete objects in JSON or YAML, respectively

Pods

- ❖ Pod Details
  - ➢ kubectl describe pods kuard
- ❖ Deleting a Pod
  - ➢ kubectl delete pods/kuard

    or using the same file that we used to create it:
  - ➢ kubectl delete -f kuard-pod.yaml
- ❖ When Pod is deleted, its not immediately killed
- ❖ in termination grace period - by default, this is 30 seconds
- ❖ no longer receives new requests - important for reliability because it allows

  the Pod to finish any active requests

Pods

- Accessing our Pod
- Using Port Forwarding
    - kubectl port-forward kuard 8080:8080
    - a secure tunnel is created from our local machine, through the Kubernetes master, to the instance of the Pod running on one of the worker nodes
    - As long as the port-forward command is still running, we can access the Pod (in this case the kuard web interface) on http://localhost:8080

Pods

- Getting More Info with Logs
  - kubectl logs kuard
  - Adding the -f flag will cause we to continuously stream logs
- Running Commands in our Container with exec
  - kubectl exec kuard date
- get an interactive session by adding the -it flags:
  - kubectl exec -it kuard ash
- Copying Files to and from Containers
  - kubectl cp <pod-name>:/captures/capture3.txt ./capture3.txt

Pods

- copy files from our local machine into a container
  - kubectl cp $HOME/config.txt <pod-name>:/config.txt
- copying files into a container is an antipattern
- we really should treat the contents of a container as immutable
- occasionally it’s the most immediate way to stop the bleeding and restore our service to health, since it is quicker than building, pushing, and rolling out a new image
- Once the bleeding is stopped, however, it is critically important that we immediately go and do the image build and rollout, or we are guaranteed to forget the local change that we made to our container and overwrite it in the subsequent regularly scheduled rollout

Pods

- Health Checks
  - application is automatically kept alive
  - ensures that the main process of our application is always running
  - if our process has deadlocked and is unable to serve requests
  - process health check still believes application is healthy since its process is still running
  - Kubernetes introduced health checks for application liveness
  - Liveness health checks
    - run application-specific logic(loading a web page)
    - are application-specific, hence define them in our Pod manifest

Pods

- Liveness Probe
  - defined per container, which means each container inside a Pod is health-checked separately

Pods

kuard-pod-health.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: hellodk/kuard-amd64:1
      name: kuard
      livenessProbe:
        httpGet:
          path: /healthy
          port: 8080
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Pods

- our Pod manifest uses an httpGet probe to perform an HTTP GET request against the /healthy endpoint on port 8080 of the kuard container
- probe sets an initialDelaySeconds of 5, and thus will not be called until five seconds after all the containers in the Pod are created
- The probe must respond within the one-second timeout, and the HTTP status code must be equal to or greater than 200 and less than 400 to be considered successful
- Kubernetes will call the probe every 10 seconds
- If more than three probes fail, the container will fail and restart

Pods

- we can see this in action by looking at the kuard status page
- Create a Pod using this manifest and then port-forward to that Pod:
  - kubectl apply -f kuard-pod-health.yaml
  - kubectl port-forward kuard 8080:8080
- Point our browser to http://localhost:8080
- Click the “Liveness Probe” tab
  - see a table that lists all of the probes that this instance of kuard has received
  - click the “fail” link on that page, kuard will start to fail health checks
  - Wait long enough and Kubernetes will restart the container

Pods

- Details of the restart can be found with
  - kubectl describe pods kuard
- Events section will have text similar to the following:
  - Killing container with id docker://2ac946...:pod

    "kuard_default(9ee84...)"

    container "kuard" is unhealthy, it will be killed and re-created

Pods

- Readiness Probe
  - Kubernetes makes a distinction between liveness and readiness
  - Liveness determines if an application is running properly
  - Containers that fail liveness checks are restarted
  - Readiness describes when a container is ready to serve user requests
  - Containers that fail readiness checks are removed from service load balancers
  - Readiness probes are configured similarly to liveness probes
- Combining the readiness and liveness probes helps ensure only healthy containers are running within the cluster

Pods

- Types of Health Checks
  - HTTP checks
  - tcpSocket health checks that open a TCP socket(telnet ip port)
  - Kubernetes allows exec probes(script with 0 as exit status)

Pods

- Resource Management
- Kubernetes allows users to specify two different resource metrics
- Resource requests specify the minimum amount of a resource required to run the application
- Resource limits specify the maximum amount of a resource that an application can consume

Pods

- Resource Requests: Minimum Required Resources
- With Kubernetes, a Pod requests the resources required to run its containers
- Kubernetes guarantees that these resources are available to the Pod
- most commonly requested resources are CPU and memory
- Kubernetes has support for other resource types as well, such as GPUs etc.
- Resources are requested per container, not per Pod
- The total resources requested by the Pod is the sum of all resources requested by all containers in the Pod
- different containers have very different CPU requirements

Pods

- request that the kuard container lands on a machine with half a CPU free and gets 128 MB of memory allocated to it, define the Pod like below:

kuard-pod-resreq.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Pods

- REQUEST LIMIT DETAILS
  - Requests used when scheduling Pods to nodes
  - scheduler will ensure that the sum of all requests of all Pods on a node does not exceed the capacity of the node
  - a Pod is guaranteed to have at least the requested resources when running on the node
  - Importantly, “request” specifies a minimum
  - does not specify a maximum cap on the resources a Pod may use

Pods

- if we have container whose code attempts to use all available CPU cores
- we create a Pod with this container that requests 0.5 CPU
- Kubernetes schedules this Pod onto a machine with a total of 2 CPU cores
- As long as it is the only Pod on the machine, it will consume all 2.0 of the available cores, despite only requesting 0.5 CPU
- If a second Pod with the same container and the same request of 0.5 CPU lands on the machine, then each Pod will receive 1.0 cores
- If a third identical Pod is scheduled, each Pod will receive 0.66 cores
- Finally, if a fourth identical Pod is scheduled, each Pod will receive the 0.5 core it requested, and the node will be at capacity

Pods

- CPU requests are implemented using the cpu-shares functionality in the Linux kernel(steal time)
- Memory requests handled similarly to CPU with an important difference
- If a container is over its memory request, the OS can't just remove memory from the process, because it's been allocated
- Consequently, when the system runs out of memory, the kubelet terminates containers whose memory usage is greater than their requested memory
- containers are automatically restarted, but with less available memory on the machine for the container to consume

Pods

- resource requests guarantee resource availability to a Pod
- critical to ensuring that containers have sufficient resources for functioning
- Capping Resource Usage with Limits
  - set a maximum resource limit usage on a Pod via resource limits
- next example, we add a limit on the pod
- establish limits on a container - kernel is configured to ensure that consumption cannot exceed these limits
- container with a CPU limit of 0.5 cores will only ever get 0.5 cores, even if the CPU is otherwise idle
- container with a memory limit of 256 MB will not be allowed additional memory (e.g., malloc will fail) if its memory usage exceeds 256 MB

Pods

kuard-pod-reslim.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Pods

- Persisting Data with Volumes
  - When a Pod is deleted or a container restarts, any and all data in the container's filesystem is also deleted
  - Kubernetes models persistent storage
- Using Volumes with Pods
- add a volume to a Pod manifest, two new stanzas to add in configuration
  - new spec.volumes section
    - defines all of the volumes that may be accessed by containers in the Pod manifest
    - not all containers are required to mount all volumes defined in the Pod

Pods

- ➢ volumeMounts array in the container definition
  - ■ defines the volumes that are mounted into a particular container, and the path where each volume should be mounted
  - ■ two different containers in a Pod can mount the same volume at different mount paths
- ❖ nest manifest defines a single new volume named kuard-data, which the kuard container mounts to the /data path

Pods

❖ kuard-pod-vol.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      hostPath:
        path: "/var/lib/kuard"
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      volumeMounts:
        - mountPath: "/data"
          name: "kuard-data"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Labels

- ❖ Labels are key/value pairs that can be attached to Kubernetes objects such as Pods and ReplicaSets
- ❖ can be arbitrary, and are useful for attaching identifying information to Kubernetes objects
- ❖ Labels provide the foundation for grouping objects

Labels

- Applying Labels
- create a few deployments (a way to create an array of Pods) with some interesting labels
- we'll use two apps(alpaca and bandicoot)with two environments for each
- along with two different versions
- create the alpaca-prod deployment and set the ver, app, and env labels:
  - kubectl run alpaca-prod --image=gcr.io/kuar-demo/kuard-amd64:1 --replicas=2 --labels="ver=1,app=alpaca,env=prod"

Labels

- Applying Labels
- create alpaca-test deployment, set the ver, app, and env labels:
  - kubectl run alpaca-test --image=gcr.io/kuar-demo/kuard-amd64:2

    --replicas=1 --labels="ver=2,app=alpaca,env=test"
- two deployments for bandicoot, name the environments prod & staging:
  - kubectl run bandicoot-prod --image=gcr.io/kuar-demo/kuard-amd64:2

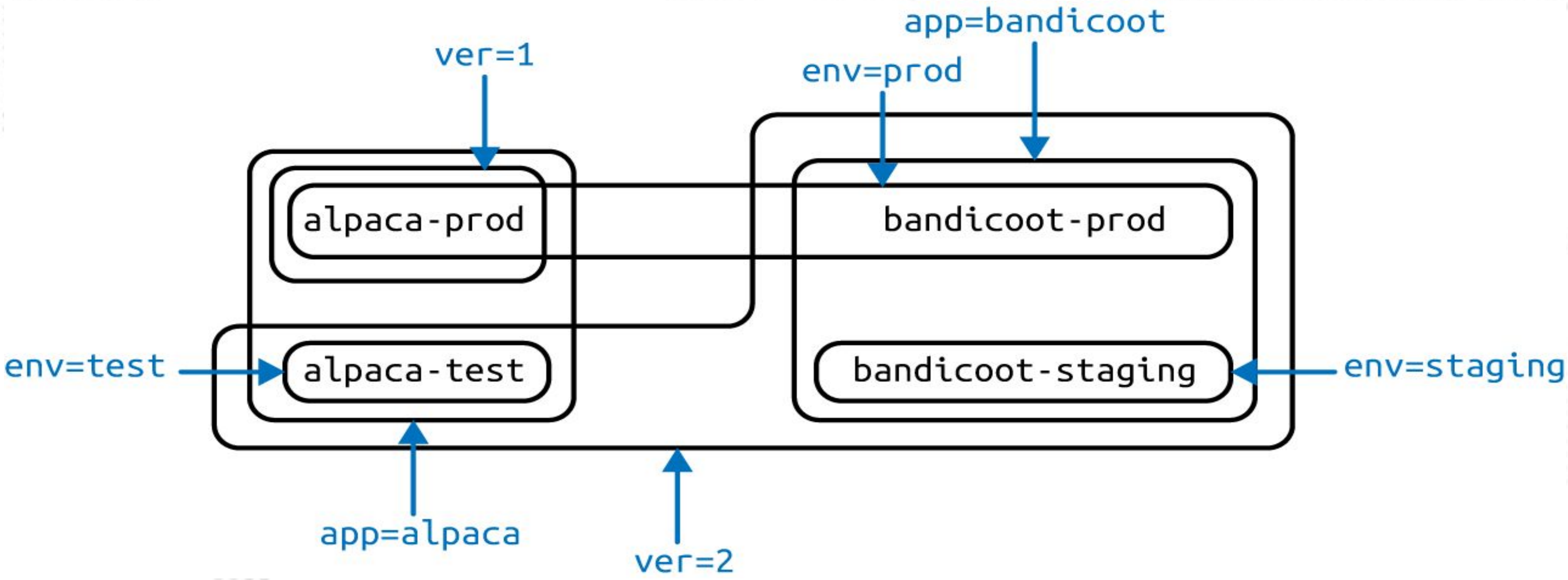
    --replicas=2 --labels="ver=2,app=bandicoot,env=prod"
  - kubectl run bandicoot-staging

    --image=gcr.io/kuar-demo/kuard-amd64:2 --replicas=1

    --labels="ver=2,app=bandicoot,env=staging"

Labels

- we have four deployments—alpaca-prod, alpaca-test, bandicoot-prod, and bandicoot-staging:
  - kubectl get deployments --show-labels



Labels

- Modifying Labels
- Labels can also be applied (or updated) on objects after they are created.
  - kubectl label deployments alpaca-test "canary=true"
- use the -L option to kubectl get to show a label value as a column:
  - kubectl get deployments -L canary
- remove a label by applying a dash suffix:
  - kubectl label deployments alpaca-test "canary-"

Labels

- Label Selectors
  - used to filter Kubernetes objects based on a set of labels
  - Selectors use a simple Boolean language
- Running the kubectl get pods command should return all the Pods currently running in the cluster
  - kubectl get pods --show-labels
- want to list pods that had the ver label set to 2?
  - kubectl get pods --selector="ver=2"
  - kubectl get pods --selector="ver!=2"

Labels

- Label Selectors
- specify two selectors separated by a comma, only the objects that satisfy both will be returned, logical AND operation:
  - kubectl get pods --selector="app=bandicoot,ver=2"
- all pods where the app label is set to alpaca or bandicoot
  - kubectl get pods --selector="app in (alpaca,bandicoot)"
- if a label is set at all
  - kubectl get deployments --selector="canary"

Labels

- A selector of app=alpaca,ver in (1, 2) would be converted to this:

```yaml
selector:
  matchLabels:
    app: alpaca
  matchExpressions:
    - {key: ver, operator: In, values: [1, 2]}
```

- Compact YAML syntax
- The selector app=alpaca,ver=1 would be represented like this:

```yaml
selector:
  app: alpaca
  ver: 1
```

Labels

- It is easy to clean up all of the deployments that we started:
  - kubectl delete deployments --all

- ✓ Pods represent the atomic unit of work in a Kubernetes cluster
- ✓ Pods are comprised of one or more containers working together symbiotically
- ✓ To create a Pod, we write a Pod manifest and submit it to the Kubernetes API server by using the command-line tool
- ✓ Once we’ve submitted the manifest to the API server, the Kubernetes scheduler finds a machine where the Pod can fit and schedules the Pod to that machine
- ✓ Once scheduled, the kubelet daemon on that machine is responsible for creating the containers that correspond to the Pod, as well as performing any health checks defined in the Pod manifest

- ✓ Once a Pod is scheduled to a node, no rescheduling occurs if that node fails
- ✓ to create multiple replicas of the same Pod we have to create and name them manually
- ✓ Labels are used to identify and optionally group objects in a Kubernetes cluster
- ✓ Labels are also used in selector queries to provide flexible runtime grouping of objects such as pods


Break

DAY: 3

Day 3

- ❖ Service Discovery
  - ➢ What Is Service Discovery?
  - ➢ The Service Object
  - ➢ Looking Beyond the Cluster
  - ➢ Cloud Integration
  - ➢ Advanced Details
  - ➢ Summary

Service Discovery

- Kubernetes is a very dynamic system
- placing Pods on nodes, making sure they are up and running, and rescheduling them as needed
- ways to automatically change the number of pods based on load
- dynamic nature of Kubernetes makes it easy to run a lot of things
- it creates problems when it comes to finding those things
- traditional network infrastructure wasn't built for the level of dynamism that Kubernetes presents

What Is Service Discovery?

- ❖ general name for this class of problems and solutions is service discovery
- ❖ solves the problem of finding which processes are listening at which addresses for which services
- ❖ A good system is also low-latency
- ❖ clients updated soon after the information associated with a service changes

What Is Service Discovery?

- Domain Name System (DNS) is the traditional system of service discovery
- designed for relatively stable name resolution with wide and efficient caching
- great system for the internet but falls short in the dynamic world
- Unfortunately, many systems (for example, Java, by default) look up a name in DNS directly and never re-resolve
- leads to clients caching stale mappings and talking to the wrong IP
- Even with short TTLs and well-behaved clients, there is a natural delay between when a name resolution changes and the client notices

The Service Object

- kubectl run command is an easy way to create a Kubernetes deployment
- use kubectl expose to create a service
- Let’s create some deployments and services so we can see how they work:
  - kubectl run alpaca-prod \

    --image=gcr.io/kuar-demo/kuard-amd64:1 \

    --replicas=3 \

    --port=8080 \

    --labels="ver=1,app=alpaca,env=prod"

The Service Object

- kubectl expose deployment alpaca-prod
- kubectl run bandicoot-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:2 \
  --replicas=2 \
  --port=8080 \
  --labels="ver=2,app=bandicoot,env=prod"
- kubectl expose deployment bandicoot-prod
- kubectl get services -o wide

NAME	CLUSTER-IP	... PORT(S)	... SELECTOR
alpaca-prod	10.115.245.13	... 8080/TCP	... app=alpaca,env=prod,ver=1
bandicoot-prod	10.115.242.3	... 8080/TCP	... app=bandicoot,env=prod,ver=2
kubernetes	10.115.240.1	... 443/TCP	... <none>

The Service Object

- service is assigned a new type of virtual IP called a cluster IP
- special IP address the system will load-balance across all of the pods internally
- interact with services, port-forward to one of the alpaca pods
- Start and leave this command running in a terminal window
- the port forward working by accessing the alpaca pod at http://localhost:48858:
  - ALPACA_POD=$(kubectl get pods -l app=alpaca \
    -o jsonpath='{.items[0].metadata.name}')
  - kubectl port-forward $ALPACA_POD 48858:8080

Service DNS

- Kubernetes provides a DNS service exposed to Pods running in the cluster
- DNS service was installed as a system component when the cluster was first created
- Kubernetes DNS service provides DNS names for cluster IPs

Service DNS

- try this out by expanding the “DNS Query” section on the kuard server status page
- Query the A record for alpaca-prod
- The output should look something like this:
  - ;; opcode: QUERY, status: NOERROR, id: 12071
    ;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
    ;; QUESTION SECTION:
    ;alpaca-prod.default.svc.cluster.local. IN A
    ;; ANSWER SECTION:
    alpaca-prod.default.svc.cluster.local. 30 IN A 10.115.245.13

Service DNS

- The full DNS name here is alpaca-prod.default.svc.cluster.local
- Let's break this down:
  - alpaca-prod
    - name of the service in question
  - default
    - namespace that this service is in
  - svc
    - Recognizing that this is a service
  - cluster.local.
    - The base domain name for the cluster

Readiness Checks

- when an application first starts up it isn't ready to handle requests
- some amount of initialization that can take anywhere from under a second to several minutes
- modify our deployment to add a readiness check:
  - kubectl edit deployment/alpaca-prod
- fetch current version of the alpaca-prod deployment & bring it up in an editor
- save and quit our editor, it'll then write the object back to Kubernetes

Readiness Checks

- quick way to edit an object without saving it to a YAML file
- Add the following section:
  - ```
    spec:
      ...
      template:
        ...
        spec:
          containers:
            ...
            name: alpaca-prod
            readinessProbe:
              httpGet:
                path: /ready
                port: 8080
              periodSeconds: 2
              initialDelaySeconds: 0
              failureThreshold: 3
              successThreshold: 1
    ```
    ```

Readiness Checks

- sets up the pods this deployment will create so that they will be checked for readiness via an HTTP GET to /ready on port 8080
- check is done every 2 seconds starting as soon as the pod comes up
- three successive checks fail, then the pod will be considered not ready
- if only one check succeeds, then the pod will again be considered ready
- Only ready pods are sent traffic
- Updating the deployment definition like this will delete and recreate the alpaca pods

Readiness Checks

- As such, we need to restart our port-forward command from earlier:
  - ALPACA_POD=$(kubectl get pods -l app=alpaca \

    -o jsonpath='{.items[0].metadata.name}')
  - kubectl port-forward $ALPACA_POD 48858:8080
- Open our browser to http://localhost:48858 and we should see the debug page for that instance of kuard
- Expand the “Readiness Probe” section
- page updates every time there is a new readiness check(every 2 seconds)
- another terminal window, start a watch command on the endpoints for the alpaca-prod service

Readiness Checks

- --watch option here causes the kubectl command to hang around and output any updates.
- easy way to see how a Kubernetes object changes over time:
  - kubectl get endpoints alpaca-prod --watch
- go back to our browser and hit the “Fail” link for the readiness check
- see that the server is now returning 500s.
- After three of these this server is removed from the list of endpoints for the service

Looking Beyond the Cluster

- Oftentimes the IPs for pods are only reachable from within the cluster
- At some point, we have to allow new traffic in!
- most portable way to do this is to use a feature called NodePorts
- addition to a cluster IP, the system picks a port
- node in the cluster then forwards traffic to that port to the service
- if we can reach any node in the cluster we can contact a service

Looking Beyond the Cluster

- Try this out by modifying the alpaca-prod service:
  - kubectl edit service alpaca-prod
- Change the spec.type field to NodePort
- we can also do this when creating the service via kubectl expose by specifying --type=NodePort

Looking Beyond the Cluster

- The system will assign a new NodePort:
  - kubectl describe service alpaca-prod

```
Name:               alpaca-prod
Namespace:          default
Labels:             app=alpaca
                    env=prod
                    ver=1
Annotations:        <none>
Selector:           app=alpaca,env=prod,ver=1
Type:               NodePort
IP:                 10.115.245.13
Port:               <unset> 8080/TCP
NodePort:           <unset> 32711/TCP
Endpoints:
10.112.1.66:8080,10.112.2.104:8080,10.112.2.105:8080
Session Affinity:   None
No events.
```

Looking Beyond the Cluster

- Here we see that the system assigned port 32711 to this service
- Now we can hit any of our cluster nodes on that port to access the service.
- Each request that we send to the service will be randomly directed to one of the Pods that implement the service

Advanced Details

- Kubernetes is built to be an extensible system
- Understanding the details of how a sophisticated concept like services is implemented may help to troubleshoot or create more advanced integrations

Endpoints

- Some applications (and the system itself) want to be able to use services without using a cluster IP
- done with another type of object called Endpoints
- For every Service object, Kubernetes creates a buddy Endpoints object that contains the IP addresses for that service:
  - kubectl describe endpoints alpaca-prod

    Name: alpaca-prod

    Namespace: default

    Labels: app=alpaca env=prod ver=1

Endpoints

```
Subsets:
  Addresses:          10.112.1.54,10.112.2.84,10.112.2.85
  NotReadyAddresses:  <none>
  Ports:
    Name      Port  Protocol
    ----      ----  --------
    <unset>   8080  TCP
No events.
```

- ❖ advanced application can talk to the Kubernetes API directly to look up endpoints and call them

Endpoints

- Kubernetes API can “watch” objects and be notified as soon as they change
- hence a client can react immediately as soon as the IPs associated with a service change
- Let’s demonstrate this
- In a terminal window, start the following command and leave it running:
  - kubectl get endpoints alpaca-prod --watch
- It will output the current state of the endpoint and then “hang”:

NAME	ENDPOINTS	AGE
alpaca-prod	10.112.1.54:8080,10.112.2.84:8080,10.112.2.85:8080	1m

Endpoints

- open up another terminal window and delete and recreate the deployment backing alpaca-prod:
  - kubectl delete deployment alpaca-prod
  - kubectl run alpaca-prod \
    --image=gcr.io/kuar-demo/kuard-amd64:1 \
    --replicas=3 \
    --port=8080 \
    --labels="ver=1,app=alpaca,env=prod"
- as we deleted and re-created these pods, the output of the command reflected the most up-to-date set of IP addresses associated with the service

Endpoints

- our output will look something like this:
  - NAME ENDPOINTS AGE

  alpaca-prod 10.112.1.54:8080,10.112.2.84:8080,10.112.2.85:8080 1m

  alpaca-prod 10.112.1.54:8080,10.112.2.84:8080 1m

  alpaca-prod <none> 1m

  alpaca-prod 10.112.2.90:8080 1m

  alpaca-prod 10.112.1.57:8080,10.112.2.90:8080 1m

  alpaca-prod 10.112.0.28:8080,10.112.1.57:8080,10.112.2.90:8080 1m
- The Endpoints object is great if we are writing new code that is built to run on Kubernetes from the start
  - kubectl get endpoints alpaca-prod

Manual Service Discovery

- Kubernetes services are built on top of label selectors over pods
- we can use the Kubernetes API to do rudimentary service discovery without using a Service object at all!
- Let’s demonstrate
  - kubectl get pods -o wide --show-labels

    NAME ... IP ... LABELS

    alpaca-prod-12334-87f8h ... 10.112.1.54 ... app=alpaca,env=prod,ver=1

    alpaca-prod-12334-jssmh ... 10.112.2.84 ... app=alpaca,env=prod,ver=1

    alpaca-prod-12334-tjp56 ... 10.112.2.85 ... app=alpaca,env=prod,ver=1

    bandicoot-prod-5678-sbxzl ... 10.112.1.55 ... app=bandicoot,env=prod,ver=2

    bandicoot-prod-5678-x0dh8 ... 10.112.2.86 ... app=bandicoot,env=prod,ver=2

Manual Service Discovery

- great, but what if we have a ton of pods?
- we'll probably want to filter this based on the labels applied as part of the deployment
- Let's do that for just the alpaca app:
  - ➢ kubectl get pods -o wide --selector=app=alpaca,env=prod

  NAME ... IP ...

  alpaca-prod-3408831585-bpzdz ... 10.112.1.54 ...

  alpaca-prod-3408831585-kncwt ... 10.112.2.84 ...

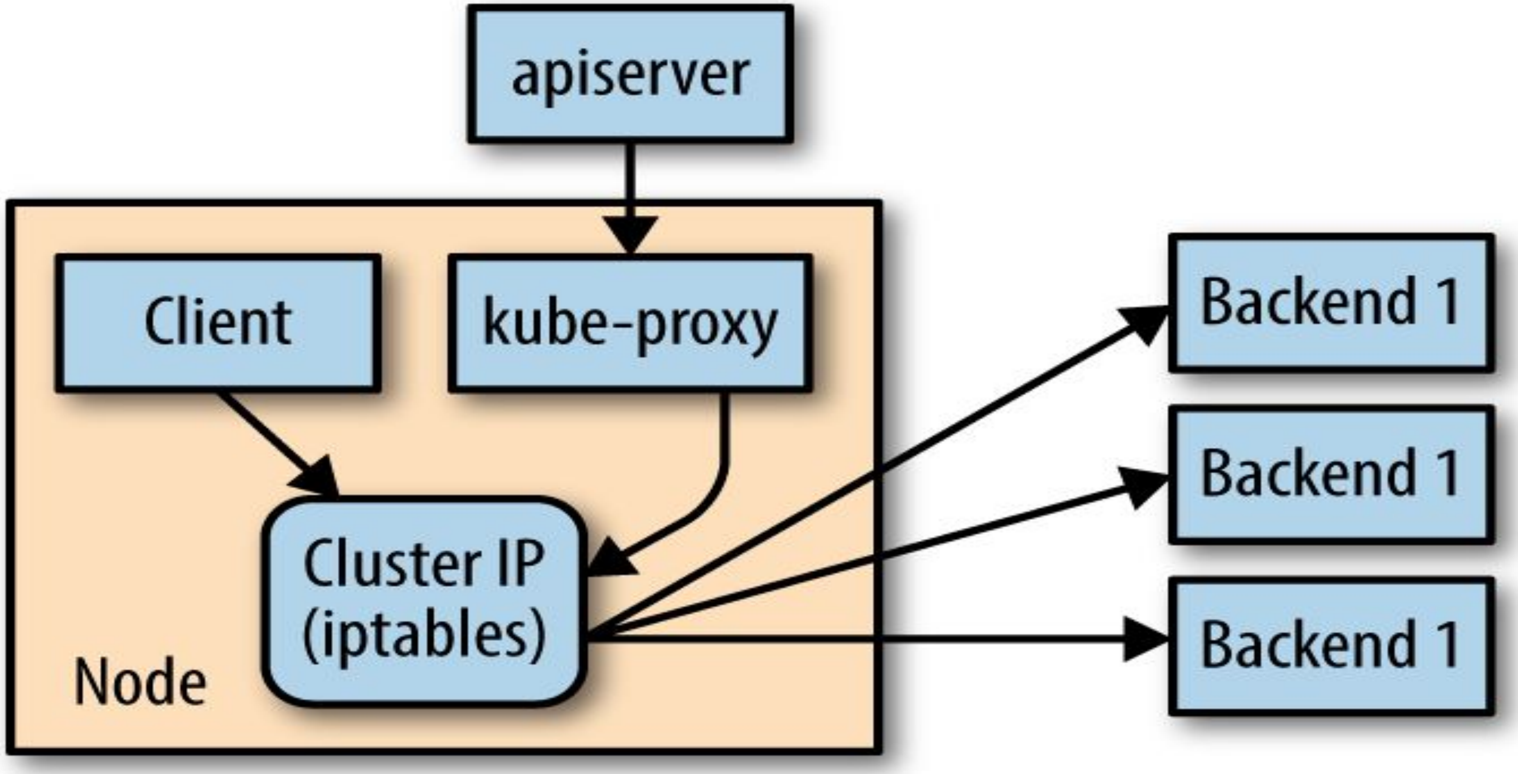
  alpaca-prod-3408831585-l9fsq ... 10.112.2.85 ...
- At this point we have the basics of service discovery!

Manual Service Discovery

- We can always use labels to identify the set of pods we are interested in, get all of the pods for those labels, and dig out the IP address
- But keeping the correct set of labels to use in sync can be tricky
- This is why the Service object was created

kube-proxy and Cluster IPs

- ❖ Cluster IPs are stable virtual IPs that load-balance traffic across all of the endpoints in a service
- ❖ This magic is performed by a component running on every node in the cluster called the kube-proxy



kube-proxy and Cluster IPs

- kube-proxy watches for new services in the cluster via the API server
- It then programs a set of iptables rules in the kernel of that host to rewrite the destination of packets so they are directed at one of the endpoints for that service
- If the set of endpoints for a service changes (due to pods coming and going or due to a failed readiness check) the set of iptables rules is rewritten
- cluster IP itself is usually assigned by the API server as the service is created
- cannot be modified without deleting and recreating the Service object

Cleanup

- ❖ Run the following commands to clean up all of the objects created in this:
  - ➢ kubectl delete services,deployments -l app

Summary

- Kubernetes is a dynamic system that challenges traditional methods of naming and connecting services over the network
- Service object provides a flexible and powerful way to expose services both within the cluster and beyond
- With the techniques covered here we can connect services to each other and expose them outside the cluster

Summary

- Once our application can dynamically find services and react to the dynamic placement of those applications, we are free to stop worrying about where things are running and when they move
- critical piece of the puzzle to start to think about services in a logical way and let Kubernetes take care of the details of container placement

Day 3

- ❖ ReplicaSets
  - ➢ Reconciliation Loops
  - ➢ Relating Pods and ReplicaSets
  - ➢ Designing with ReplicaSets
  - ➢ ReplicaSet Spec
  - ➢ Creating a ReplicaSet
  - ➢ Inspecting a ReplicaSet
  - ➢ Scaling ReplicaSets
  - ➢ Deleting ReplicaSets
  - ➢ Summary

ReplicaSets

- Previously, we covered how to run individual containers as Pods
- But these pods are essentially one-off singletons
- More often than not, we want multiple replicas of a container running at a particular time
- variety of reasons for this type of replication:
- Redundancy
  - Multiple running instances mean failure can be tolerated
- Scale
  - Multiple running instances mean that more requests can be handled

ReplicaSets

- Sharding
  - Different replicas can handle different parts of a computation in parallel
- a user managing a replicated set of Pods considers them as a single entity to be defined and managed - This is precisely what a ReplicaSet is
- A ReplicaSet acts as a cluster-wide Pod manager, ensuring that the right types and number of Pods are running at all times
- ReplicaSets make it easy to create and manage replicated sets of Pods
- Pods managed by ReplicaSets are automatically rescheduled under certain failure conditions such as node failures and network partitions

Reconciliation Loops

- The central concept behind a reconciliation loop is the notion of desired state and observed or current state
- Ex: desired state is that there are three replicas of a Pod running the kuard server
  - there are only two kuard Pods currently running
  - reconciliation loop is constantly running, observing the current state of the world and taking action to try to make the observed state match the desired state

Designing with ReplicaSets

- ReplicaSets are designed to represent a single, scalable microservice inside our architecture
- The key characteristic of ReplicaSets is that every Pod that is created by the ReplicaSet controller is entirely homogeneous
- ReplicaSets are designed for stateless (or nearly stateless) services
- The elements created by the ReplicaSet are interchangeable; when a ReplicaSet is scaled down, an arbitrary Pod is selected for deletion
- our application’s behavior shouldn’t change because of such a scale-down operation

ReplicaSet Spec

- kuard-rs.yaml
  - ```yaml
    apiVersion: extensions/v1beta1
    kind: ReplicaSet
    metadata:
        name: kuard
    spec:
        replicas: 1
        template:
            metadata:
                labels:
                    app: kuard
                    version: "2"
            spec:
                containers:
                    - name: kuard
                      image: "gcr.io/kuar-demo/kuard-amd64:2"
    ```
    ```

ReplicaSet Spec

- Pod Templates
  - when the number of Pods in the current state is less than the number of Pods in the desired state, the ReplicaSet controller will create new Pods
  - Pods are created using a Pod template that is contained in the ReplicaSet specification

ReplicaSet Spec

- Pod template in a ReplicaSet:
  - template:

```
metadata:
  labels:
    app: helloworld
    version: v1
spec:
  containers:
    - name: helloworld
      image: kelseyhightower/helloworld:v1
      ports:
        - containerPort: 80
```

ReplicaSet Spec

- Labels
  - used to filter Pod listings and track Pods running within a cluster
  - When ReplicaSets are initially created, the ReplicaSet fetches a Pod listing from the Kubernetes API and filters the results by labels
  - labels used for filtering are defined in the ReplicaSet spec section and are the key to understanding how ReplicaSets work
  - NOTE
    - The selector in the ReplicaSet spec should be a proper subset of the labels in the Pod template

Creating a ReplicaSet

- create a ReplicaSet using a configuration file and the kubectl apply
  - kubectl apply -f kuard-rs.yaml

    replicaset "kuard" created

Creating a ReplicaSet

- Once the kuard ReplicaSet has been accepted, the ReplicaSet controller will detect there are no kuard Pods running that match the desired state
- a new kuard Pod will be created based on the contents of the Pod template:
  - kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
kuard-yvzgd	1/1	Running	0	11s

Inspecting a ReplicaSet

- kubectl describe rs kuard

```
Name:         kuard
Namespace:    default
Image(s):     kuard:1.9.15
Selector:     app=kuard,version=2
Labels:       app=kuard,version=2
Replicas:     1 current / 1 desired
Pods Status:  1 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
```

Inspecting a ReplicaSet

- Finding a ReplicaSet from a Pod
    - Sometimes we may wonder if a Pod is being managed by a ReplicaSet, and, if it is, which ReplicaSet
    - run the following, look for the kubernetes.io/created-by entry in the annotations section:
        - kubectl get pods <pod-name> -o yaml

Inspecting a ReplicaSet

- ➢ this will list the name of the ReplicaSet that is managing this Pod
- ❖ Finding a Set of Pods for a ReplicaSet
  - ➢ we can also determine the set of Pods managed by a ReplicaSet
  - ➢ get the set of labels using the kubectl describe command
    - ■ the label selector was app=kuard,version=2
  - ➢ find the Pods that match this selector, use the --selector flag or the shorthand -l:
    - ■ kubectl get pods -l app=kuard,version=2

Scaling ReplicaSets

- ReplicaSets are scaled up or down by updating the spec.replicas key
- When a ReplicaSet is scaled up, new Pods are added
- Imperative Scaling with kubectl Scale
    - kubectl scale replicasets kuard --replicas=4

Scaling ReplicaSets

- scale the kuard ReplicaSet, edit the kuard-rs.yaml configuration file and set the replicas count to 3:
  - ```
    ...
    spec:
      replicas: 3
    ...
    ```
- use the kubectl apply command to submit the updated kuard ReplicaSet to the API server:
  - kubectl apply -f kuard-rs.yaml
    ```

Scaling ReplicaSets

- see output like the following:
  - kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
kuard-3a2sb	1/1	Running	0	26s
kuard-wuq9v	1/1	Running	0	26s
kuard-yvzgd	1/1	Running	0	2m

- Autoscaling a ReplicaSet
  - times when we want to have explicit control over the number of replicas in a ReplicaSet
  - often we simply want to have “enough” replicas in the Replicaset

Scaling ReplicaSets

- ➢ For example, with a web server like nginx, we may want to scale due to CPU usage
- ➢ For an in-memory cache, we may want to scale with memory consumption
- ➢ In some cases we may want to scale in response to custom application metrics
- ➢ Kubernetes can handle all of these scenarios via horizontal pod autoscaling (HPA)

Scaling ReplicaSets

- HPA requires the presence of the heapster Pod on our cluster.
- heapster keeps track of metrics and provides an API for consuming metrics HPA uses when making scaling decisions
- Most installations of Kubernetes include heapster by default
- validate its presence by listing the Pods in the kube-system namespace:
  - kubectl get pods --namespace=kube-system
- we should see a Pod named heapster somewhere in that list
- If we do not see it, autoscaling will not work correctly

Scaling ReplicaSets

- AUTOSCALING BASED ON CPU
  - most useful for request-based systems that consume CPU proportionally to the number of requests they are receiving, while using a relatively static amount of memory
  - scale a ReplicaSet, we can run a command like the following:
    - kubectl autoscale rs kuard --min=2 --max=5 --cpu-percent=80
  - creates an autoscaler that scales between two and five replicas with a CPU threshold of 80%

Scaling ReplicaSets

- horizontalpodautoscalers is quite a bit to type, can be shortened to hpa:
  - kubectl get hpa
- default, this also deletes the Pods that are managed by the ReplicaSet:
  - kubectl delete rs kuard
    replicaset "kuard" deleted

Deleting ReplicaSets

- Running the kubectl get pods command shows that all the kuard Pods created by the kuard ReplicaSet have also been deleted:
  - kubectl get pods
- don’t want to delete the Pods that are being managed by the ReplicaSet?
- set the --cascade flag to false to ensure only the ReplicaSet object is deleted and not the Pods:
  - kubectl delete rs kuard --cascade=false

Summary

- Composing Pods with ReplicaSets provides the foundation for building robust applications with automatic failover, and makes deploying those applications a breeze by enabling scalable and sane deployment patterns
- ReplicaSets should be used for any Pod we care about, even if it is a single Pod!
- Some people even default to using ReplicaSets instead of Pods
- typical cluster will have many ReplicaSets, so apply liberally to the affected area

Day 3

- DaemonSets
    - DaemonSet Scheduler
    - Creating DaemonSets
    - Limiting DaemonSets to Specific Nodes
    - Updating a DaemonSet
    - Deleting a DaemonSet
    - Summary

DaemonSets

- ReplicaSets are about creating a service (e.g., a web server) with multiple replicas for redundancy
- not the only reason we may want to replicate a set of Pods within a cluster
- Another reason to replicate a set of Pods is to schedule a single Pod on every node within the cluster
- motivation for replicating a Pod to every node is to land some sort of agent or daemon on each node, and the Kubernetes object for achieving this is the DaemonSet
- DaemonSet ensures a copy of a Pod is running across a set of nodes in a Kubernetes cluster

DaemonSets

- DaemonSets are used to deploy system daemons such as log collectors and monitoring agents, which typically must run on every node
- DaemonSets share similar functionality with ReplicaSets
  - both create Pods that are expected to be long-running services and ensure that the desired state and the observed state of the cluster match
- important to understand when to use one over the other
- ReplicaSets should be used when our application is completely decoupled from the node and we can run multiple copies on a given node without special consideration

DaemonSets

- DaemonSets should be used when a single copy of our application must run on all or a subset of the nodes in the cluster
- find ourself wanting a single Pod per node?
  - a DaemonSet is the correct Kubernetes resource to use
- if we find ourself building a homogeneous replicated service to serve user traffic, then a ReplicaSet is probably the right Kubernetes resource to use
- If a new node is added to the cluster, then the DaemonSet controller notices that it is missing a Pod and adds the Pod to the new node

DaemonSet Scheduler

- ❖ By default a DaemonSet will create a copy of a Pod on every node unless a node selector is used
- ❖ node selector limits eligible nodes to those with a matching set of labels
- ❖ DaemonSets determine which node a Pod will run on at Pod creation time by specifying the nodeName field in the Pod spec
- ❖ Pods created by DaemonSets are ignored by the Kubernetes scheduler
- ❖ Like ReplicaSets, DaemonSets are managed by a reconciliation control loop that measures the desired state (a Pod is present on all nodes) with the observed state (is the Pod present on a particular node?)

Creating DaemonSets

- DaemonSets are created by submitting a DaemonSet configuration to the Kubernetes API server
- following DaemonSet will create a fluentd logging agent on every node in the target cluster
- fluentd.yaml

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
  labels:
    app: fluentd
```

Creating DaemonSets

```yaml
spec:
 template:
  metadata:
   labels:
    app: fluentd
  spec:
   containers:
   - name: fluentd
    image: fluent/fluentd:v0.14.10
    resources:
     limits:
      memory: 200Mi
     requests:
      cpu: 100m
      memory: 200Mi
```

Creating DaemonSets

```
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
  terminationGracePeriodSeconds: 30
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: varlibdockercontainers
    hostPath:
      path: /var/lib/docker/containers
```

Creating DaemonSets

- Each DaemonSet must include a Pod template spec, which will be used to create Pods as needed.
- This is where the similarities between ReplicaSets and DaemonSets end
- Unlike ReplicaSets, DaemonSets will create Pods on every node in the cluster by default unless a node selector is used
- Once we have a valid DaemonSet configuration in place, we can use the kubectl apply command to submit the DaemonSet to the Kubernetes API
- create a DaemonSet to ensure the fluentd HTTP server is running on every node in our cluster:
    - kubectl apply -f fluentd.yaml

Creating DaemonSets

- query its current state using the kubectl describe command:
  - kubectl describe daemonset fluentd --namespace=kube-system

    Name: fluentd

    Image(s): fluent/fluentd:v0.14.10

    Selector: app=fluentd

    Node-Selector: <none>

    Labels: app=fluentd

    Desired Number of Nodes Scheduled: 3

    Current Number of Nodes Scheduled: 3

    Number of Nodes Misscheduled: 0

    Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed

Creating DaemonSets

- verify this using the kubectl get pods command with the -o flag to print the nodes where each fluentd Pod was assigned
  - kubectl get pods -o wide

NAME	AGE	NODE
fluentd-1q6c6	13m	k0-default-pool-35609c18-z7tb
fluentd-mwi7h	13m	k0-default-pool-35609c18-ydae
fluentd-zr6l7	13m	k0-default-pool-35609c18-pol3

- adding a new node to the cluster will result in a fluentd Pod being deployed to that node automatically:
  - kubectl get pods -o wide

Creating DaemonSets

- This output indicates a fluentd Pod was successfully deployed to all three nodes in our cluster.
- We can verify this using the kubectl get pods command with the -o flag to print the nodes where each fluentd Pod was assigned:
  - kubectl get pods -o wide

NAME	AGE	NODE
fluentd-1q6c6	13m	k0-default-pool-35609c18-z7tb
fluentd-mwi7h	13m	k0-default-pool-35609c18-ydae
fluentd-zr6l7	13m	k0-default-pool-35609c18-pol3

Creating DaemonSets

- With the fluentd DaemonSet in place, adding a new node to the cluster will result in a fluentd Pod being deployed to that node automatically:
  - kubectl get pods -o wide

NAME	AGE	NODE
fluentd-1q6c6	13m	k0-default-pool-35609c18-z7tb
fluentd-mwi7h	13m	k0-default-pool-35609c18-ydae
fluentd-oipmq	43s	k0-default-pool-35609c18-0xnl
fluentd-zr6l7	13m	k0-default-pool-35609c18-pol3

- this is how the Kubernetes DaemonSet controller reconciles its observed state with our desired state

Limiting DaemonSets to Specific Nodes

- cases where we want to deploy a Pod to only a subset of nodes
- For example, maybe we have a workload that requires a GPU or access to fast storage only available on a subset of nodes in our cluster
- In cases like these node labels can be used to tag specific nodes that meet workload requirements
- Adding Labels to Nodes
  - first step in limiting DaemonSets to specific nodes is to add the desired set of labels to a subset of nodes

Limiting DaemonSets to Specific Nodes

- following command adds the ssd=true label to a single node:
  - kubectl label nodes k0-default-pool-35609c18-z7tb ssd=true

    node "k0-default-pool-35609c18-z7tb" labeled
- Just like with other Kubernetes resources, listing nodes without a label selector returns all nodes in the cluster:
  - kubectl get nodes

NAME	STATUS	AGE
k0-default-pool-35609c18-0xnl	Ready	23m
k0-default-pool-35609c18-pol3	Ready	1d
k0-default-pool-35609c18-ydae	Ready	1d
k0-default-pool-35609c18-z7tb	Ready	1d

Limiting DaemonSets to Specific Nodes

- ➢ Using a label selector we can filter nodes based on labels
- ➢ To list only the nodes that have the ssd label set to true, use the kubectl get nodes command with the --selector flag:
  - kubectl get nodes --selector ssd=true

NAME	STATUS	AGE
k0-default-pool-35609c18-z7tb	Ready	1d

- ❖ Node Selectors
  - ➢ Node selectors can be used to limit what nodes a Pod can run on in a given Kubernetes cluster
  - ➢ Node selectors are defined as part of the Pod spec when creating a DaemonSet

Limiting DaemonSets to Specific Nodes

- following DaemonSet configuration limits nginx to running only on nodes with the ssd=true label
- nginx-fast-storage.yaml
  - ```yaml
    apiVersion: extensions/v1beta1
    kind: "DaemonSet"
    metadata:
      labels:
        app: nginx
        ssd: "true"
      name: nginx-fast-storage
    ```
    ```

Limiting DaemonSets to Specific Nodes

```yaml
spec:
  template:
    metadata:
      labels:
        app: nginx
        ssd: "true"
    spec:
      nodeSelector:
        ssd: "true"
      containers:
        - name: nginx
          image: nginx:1.10.0
```

Limiting DaemonSets to Specific Nodes

- Let's see what happens when we submit the nginx-fast-storage DaemonSet to the Kubernetes API:
  - kubectl apply -f nginx-fast-storage.yaml
- there is only one node with the ssd=true label, the nginx-fast-storage Pod will only run on that node:
  - kubectl get pods -o wide

NAME	STATUS	NODE
nginx-fast-storage-7b90t	Running	k0-default-pool-35609c18-z7tb

- Adding the ssd=true label to additional nodes will cause the nginx-fast-storage Pod to be deployed on those nodes

Limiting DaemonSets to Specific Nodes

- ➢ inverse is also true: if a required label is removed from a node, the Pod will be removed by the DaemonSet controller
- ➢ WARNING
  - Removing labels from a node that are required by a DaemonSet's node selector will cause the Pod being managed by that DaemonSet to be removed from the node

Deleting a DaemonSet

- Deleting a DaemonSet is pretty straightforward using the kubectl delete command
- Just be sure to supply the correct name of the DaemonSet we would like to delete:
  - kubectl delete -f fluentd.yaml
- WARNING
  - Deleting a DaemonSet will also delete all the Pods being managed by that DaemonSet
  - Set the --cascade flag to false to ensure only the DaemonSet is deleted and not the Pods

Summary

- DaemonSets provide an easy-to-use abstraction for running a set of Pods on every node in a Kubernetes cluster, or if the case requires it, on a subset of nodes based on labels.
- The DaemonSet provides its own controller and scheduler to ensure key services like monitoring agents are always up and running on the right nodes in our cluster.
- For some applications, we simply want to schedule a certain number of replicas; we don’t really care where they run as long as they have sufficient resources and distribution to operate reliably.

Summary

- However, there is a different class of applications, like agents and monitoring applications, that need to be present on every machine in a cluster to function properly
- These DaemonSets aren't really traditional serving applications, but rather add additional capabilities and features to the Kubernetes cluster itself.
- Because the DaemonSet is an active declarative object managed by a controller, it makes it easy to declare our intent that an agent run on every machine without explicitly placing it on every machine
- This is especially useful in the context of an autoscaled Kubernetes cluster where nodes may constantly be coming and going without user intervention

Summary

- In such cases, the DaemonSet automatically adds the proper agents to each node as it is added to the cluster by the autoscaler


Break

DAY: 4

Day 4

- ❖ Jobs
  - ➢ The Job Object
  - ➢ Job Patterns
  - ➢ Summary
- ❖ ConfigMaps and Secrets
  - ➢ ConfigMaps
  - ➢ Secrets
  - ➢ Naming Constraints
  - ➢ Managing ConfigMaps and Secrets
  - ➢ Summary

Jobs

- focused on long-running processes such as databases and web applications
- workloads run until either they are upgraded or the service is no longer needed
- there is often a need to run short-lived, one-off tasks
- Job object is made for handling these types of tasks
- Job creates Pods that run until successful termination (i.e., exit with 0)
- In contrast, a regular Pod will continually restart regardless of its exit code
- useful for things we want to do once - database migrations or batch jobs
- If it runs as a regular Pod, our database migration task would run in a loop, continually repopulating the database after every exit

The Job Object

- ❖ Job object is responsible for creating and managing pods defined in a template in the Job specification
- ❖ pods generally run until successful completion
- ❖ Job object coordinates running a number of pods in parallel
- ❖ If the Pod fails before a successful termination, the Job controller will create a new Pod based on the Pod template in the Job specification
- ❖ Given that Pods have to be scheduled, there is a chance that our Job will not execute if the required resources are not found by the scheduler
- ❖ due to the nature of distributed systems there is a small chance, during certain failure scenarios, that duplicate pods will be created for a specific task

Job Patterns

- Job patterns

Type	Use case	Behavior	completions	parallelism
One shot	Database migrations	A single pod running once until successful termination	1	1
Parallel fixed completions	Multiple pods processing a set of work in parallel	One or more Pods running one or more times until reaching a fixed completion count	1+	1+
Work queue: parallel Jobs	Multiple pods processing from a centralized work queue	One or more Pods running once until successful termination	1	2+

Job Patterns - One Shot

- One-shot Jobs provide a way to run a single Pod once until successful termination
- work involved in pulling this off
  - First, a Pod must be created and submitted to the Kubernetes API
  - This is done using a Pod template defined in the Job configuration
  - Once a Job is up and running, the Pod backing the Job must be monitored for successful termination
  - A Job can fail for any number of reasons including an application error, an uncaught exception during runtime, or a node failure before the Job has a chance to complete

Job Patterns - One Shot

- Job controller is responsible for recreating the Pod until a successful termination occurs
  - kubectl run -i oneshot \

    --image=gcr.io/kuar-demo/kuard-amd64:1 \

    --restart=OnFailure \

    -- --keygen-enable \

    --keygen-exit-on-complete \

    --keygen-num-to-gen 10

Job Patterns - One Shot

- After the Job has completed, the Job object and related Pod are still around
- so that we can inspect the log output
- Job won’t show up in kubectl get jobs unless we pass the -a flag
  - kubectl logs oneshot-4kfdt
- Without this flag kubectl hides completed Jobs
- Delete the Job before continuing:
  - kubectl delete jobs oneshot

Job Patterns - One Shot

- Create the same job via yaml file
  - job-oneshot.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: oneshot
  labels:
    item: jobs
spec:
  template:
    metadata:
      labels:
        item: jobs
    spec:
      containers:
      - name: kuard
        image: gcr.io/kuar-demo/kuard-amd64:1
        imagePullPolicy: Always
        args:
        - "--keygen-enable"
        - "--keygen-exit-on-complete"
        - "--keygen-num-to-gen=10"
      restartPolicy: OnFailure
```

Job Patterns – One Shot

- kubectl apply -f job-oneshot.yaml
- kubectl describe jobs oneshot
- kubectl logs oneshot-4kfdt

Job Patterns – One Shot

- kubectl apply -f job-oneshot.yaml
- kubectl describe jobs oneshot
- kubectl logs oneshot-4kfdt

POD FAILURE

- We just saw how a Job can complete successfully.
- But what happens if something fails?
- Let’s try that out and see what happens.
- Let’s modify the arguments to kuard in our configuration file to cause it to fail out with a nonzero exit code after generating three keys

POD FAILURE

- ob-oneshot-failure1.yaml

```
...
spec:
  template:
    spec:
      containers:
        ...
        args:
        - "--keygen-enable"
        - "--keygen-exit-on-complete"
        - "--keygen-exit-code=1"
        - "--keygen-num-to-gen=3"
...
```

POD FAILURE

- kubectl apply -f job-oneshot-failure1.yaml.
- Let it run for a bit and then look at the pod status:
  - kubectl get pod -a -l job-name=oneshot

NAME	READY	STATUS	RESTARTS	AGE
oneshot-3ddk0	0/1	CrashLoopBackOff	4	3m

- Here we see that the same Pod has restarted four times
- Kubernetes is in CrashLoopBackOff for this Pod
- not uncommon to have a bug someplace that causes a program to crash as soon as it starts
- Kubernetes will wait a bit before restarting the pod to avoid a crash loop eating resources on the node

POD FAILURE

- This is all handled local to the node by the kubelet without the Job being involved at all
- Kill the Job (kubectl delete jobs oneshot), and let’s try something else
- Modify the config file again and change the restartPolicy from OnFailure to Never
- Launch this with kubectl apply -f jobs-oneshot-failure2.yaml

POD FAILURE

- If we let this run for a bit and then look at related pods we'll find something interesting:
  - kubectl get pod -l job-name=oneshot -a

    NAME READY STATUS RESTARTS AGE oneshot-0wm49 0/1 Error 0 1m oneshot-6h9s2 0/1 Error 0 39s oneshot-hkzw0 1/1 Running 0 6s oneshot-k5swz 0/1 Error 0 28s oneshot-m1rdw 0/1 Error 0 19s oneshot-x157b 0/1 Error 0 57s
- What we see is that we have multiple pods here that have errored out
- By setting restartPolicy: Never we are telling the kubelet not to restart the Pod on failure, but rather just declare the Pod as failed
- The Job object then notices and creates a replacement Pod

POD FAILURE

- If we aren't careful, this'll create a lot of "junk" in our cluster
- For this reason, we suggest we use restartPolicy: OnFailure so failed Pods are rerun in place
- Clean this up with
  - kubectl delete jobs oneshot
- So far we've seen a program fail by exiting with a nonzero exit code
- But workers can fail in other ways
- Specifically, they can get stuck and not make any forward progress
- To help cover this case, we can use liveness probes with Jobs
- If the liveness probe policy determines that a Pod is dead, it'll be restarted/replaced for we

Job Patterns - Parallelism

- Generating keys can be slow
- Let's start a bunch of workers together to make key generation faster
- use a combination of the completions and parallelism parameters
- Our goal is to generate 100 keys by having 10 runs of kuard with each run generating 10 keys

Job Patterns - Parallelism

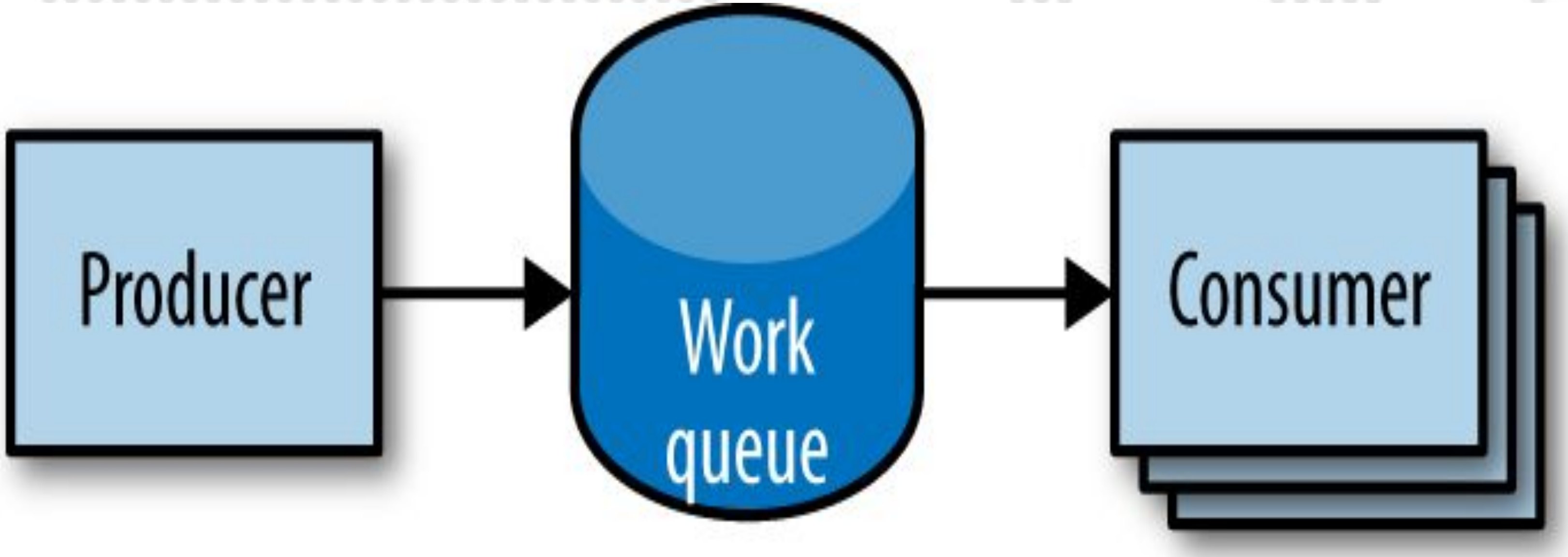
- job-parallel.yaml
  - ```yaml
    apiVersion: batch/v1
    kind: Job
    metadata:
      name: parallel
      labels:
        item: jobs
    spec:
      parallelism: 5
      completions: 10
      template:
        metadata:
          labels:
            item: jobs
        spec:
          containers:
          - name: kuard
            image: gcr.io/kuar-demo/kuard-amd64:1
            imagePullPolicy: Always
            args:
            - "--keygen-enable"
            - "--keygen-exit-on-complete"
            - "--keygen-num-to-gen=10"
          restartPolicy: OnFailure
    ```
    ```

Job Patterns - Parallelism

- Start it up:
  - kubectl apply -f job-parallel.yaml
- Now watch as the pods come up, do their thing, and exit
- New pods are created until 10 have completed altogether
- Here we use the --watch flag to have kubectl stay around and list changes as they happen:
  - kubectl get pods -w
- Feel free to poke around at the completed Jobs and check out their logs to see the fingerprints of the keys they generated
- Clean up by deleting the finished Job object with kubectl delete job parallel

Work Queues

- common use case for Jobs is to process work from a work queue
- some task creates a number of work items and publishes them to a work queue
- A worker Job can be run to process each work item until the work queue is empty



STARTING A WORK QUEUE

- start by launching a centralized work queue service
- kuard has a simple memory-based work queue system built in
- We will start an instance of kuard to act as a coordinator for all the work to be done
- Create a simple ReplicaSet to manage a singleton work queue daemon
- We are using a ReplicaSet to ensure that a new Pod will get created in the face of machine failure

STARTING A WORK QUEUE

- rs-queue.yaml

  - ```yaml
    apiVersion: extensions/v1beta1
    kind: ReplicaSet
    metadata:
     labels:
      app: work-queue
      component: queue
      item: jobs
     name: queue
    spec:
     replicas: 1
     template:
      metadata:
       labels:
        app: work-queue
        component: queue
        item: jobs
      spec:
       containers:
       - name: queue
        image: "gcr.io/kuar-demo/kuard-amd64:1"
        imagePullPolicy: Always
    ```
    ```

STARTING A WORK QUEUE

- ❖ Run the work queue with the following command:
  - ➢ kubectl apply -f rs-queue.yaml
- ❖ At this point the work queue daemon should be up and running
- ❖ Let’s use port forwarding to connect to it
- ❖ Leave this command running in a terminal window:
  - ➢ QUEUE_POD=$(kubectl get pods -l

    app=work-queue,component=queue -o

    jsonpath='{.items[0].metadata.name}')
  - ➢ kubectl port-forward $QUEUE_POD 8080:8080

STARTING A WORK QUEUE

- browse to http://localhost:8080 and see the kuard interface
- Switch to the "MemQ Server" tab to keep an eye on what is going on
- With the work queue server in place, we should expose it using a service
- This will make it easy for producers and consumers to locate the work queue via DNS

STARTING A WORK QUEUE

- service-queue.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
 labels:
  app: work-queue
  component: queue
  item: jobs
 name: queue
spec:
 ports:
 - port: 8080
  protocol: TCP
  targetPort: 8080
 selector:
  app: work-queue
  component: queue
```

- Create the queue service with kubectl:
  - kubectl apply -f service-queue.yaml

LOADING UP THE QUEUE

- We are now ready to put a bunch of work items in the queue
- use curl to drive the API for the work queue server and insert a bunch of work items
- curl will communicate to the work queue through the kubectl port-forward
- load-queue.sh
  - Create a work queue called 'keygen'
    - curl -X PUT localhost:8080/memq/server/queues/keygen
  - Create 100 work items and load up the queue
    - for i in work-item-{0..99}; do
      curl -X POST
      localhost:8080/memq/server/queues/keygen/enqueue -d "$i"
      done

LOADING UP THE QUEUE

- Run these commands, and we should see 100 JSON objects output to our terminal with a unique message identifier for each work item
- confirm the status of the queue by looking at the "MemQ Server" tab in the UI, or we can ask the work queue API directly:
  - curl 127.0.0.1:8080/memq/server/stats

```
{
  "kind": "stats",
  "queues": [
    {
      "depth": 100,
      "dequeued": 0,
      "drained": 0,
      "enqueued": 100,
      "name": "keygen"
    }
  ]
}
```

CREATING THE CONSUMER JOB

- This is where things get interesting!
- kuard is also able to act in consumer modex
- set it up to draw work items from the work queue, create a key, and then exit once the queue is empty
- job-consumers.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
    labels:
        app: message-queue
        component: consumer
        item: jobs
    name: consumers
```

CREATING THE CONSUMER JOB

```yaml
spec:
  parallelism: 5
  template:
    metadata:
      labels:
        app: message-queue
        component: consumer
        item: jobs
    spec:
      containers:
      - name: worker
        image: "gcr.io/kuar-demo/kuard-amd64:1"
        imagePullPolicy: Always
        args:
        - "--keygen-enable"
        - "--keygen-exit-on-complete"
        - "--keygen-memq-server=http://queue:8080/memq/server"
        - "--keygen-memq-queue=keygen"
      restartPolicy: OnFailure
```

CREATING THE CONSUMER JOB

- We are telling the Job to start up five pods in parallel.
- the completions parameter is unset, we put the Job into a worker pool mode
- Once the first pod exits with a zero exit code, the Job will start winding down and will not start any new Pods
- none of the workers should exit until the work is done and they are all in the process of finishing up
- Create the consumers Job:
  - kubectl apply -f job-consumers.yaml

CREATING THE CONSUMER JOB

- view the pods backing the Job:
  - kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
queue-43s87	1/1	Running	0	5m
consumers-6wjxc	1/1	Running	0	2m
consumers-7l5mh	1/1	Running	0	2m
consumers-hvz42	1/1	Running	0	2m
consumers-pc8hr	1/1	Running	0	2m
consumers-w20cc	1/1	Running	0	2m

- five pods running in parallel & will continue to run until work queue is empty
- watch as it happens in the UI on the work queue server
- As the queue empties, the consumer pods will exit cleanly and the consumers Job will be considered complete

CLEANING UP

- Using labels we can clean up all of the stuff we created in this section:
  - kubectl delete rs,svc,job -l item=jobs

Summary

- On a single cluster, Kubernetes can handle both long-running workloads such as web applications and short-lived workloads such as batch jobs
- Job abstraction allows we to model batch job patterns ranging from simple one-time tasks to parallel jobs that process many items until work has been exhausted
- Jobs are a low-level primitive and can be used directly for simple workloads
- However, Kubernetes is built from the ground up to be extensible by higher-level objects
- Jobs are no exception; they can easily be used by higher-level orchestration systems to take on more complex tasks

Day 4

- ❖ ConfigMaps & Secrets
  - ➢ ConfigMaps
  - ➢ Secrets
  - ➢ Naming Constraints
  - ➢ Managing

ConfigMaps & Secrets

- good practice to make container images as reusable as possible
- same image should be able to be used for development, staging, production
- even better if the same image is general purpose enough to be used across applications and services
- Testing and versioning get riskier and more complicated if images need to be recreated for each new environment
- But then how do we specialize the use of that image at runtime?
- This is where ConfigMaps and secrets come into play

ConfigMaps & Secrets

- ConfigMaps are used to provide configuration information for workloads
- can either be fine-grained information (a short string) or a composite value in the form of a file
- Secrets are similar to ConfigMaps but focused on making sensitive information available to the workload
- can be used for things like credentials or TLS certificates

ConfigMaps & Secrets

- ConfigMaps
- think of ConfigMap is as a Kubernetes object that defines a small filesystem
- Another way - as a set of variables that can be used when defining the environment or command line for our containers
- key thing is that the ConfigMap is combined with the Pod right before it is run
- means container image and the pod definition itself can be reused across many apps by just changing the ConfigMap that is used

ConfigMaps & Secrets

- Creating ConfigMaps
  - my-config.txt

```
# This is a sample config file that I might use to configure an application

parameter1 = value1

parameter2 = value2
```

- kubectl create configmap my-config --from-file=my-config.txt

  --from-literal=extra-param=extra-value

  --from-literal=another-param=another-value

- kubectl get configmaps my-config -o yaml

ConfigMaps & Secrets

- Using a ConfigMap - three main ways to use a ConfigMap:
- Filesystem
  - You can mount a ConfigMap into a Pod
  - A file is created for each entry based on the key name. The contents of that file are set to the value
- Environment variable
  - A ConfigMap can be used to dynamically set the value of an environment variable
- Command-line argument
  - Kubernetes supports dynamically creating the command line for a container based on ConfigMap values

ConfigMaps & Secrets

kuard-config.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard-config
spec:
  containers:
    - name: test-container
      image: gcr.io/kuar-demo/kuard-amd64:1
      imagePullPolicy: Always
      command:
        - "/kuard"
        - "$(EXTRA_PARAM)"
      env:
        - name: ANOTHER_PARAM
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: another-param
        - name: EXTRA_PARAM
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: extra-param
      volumeMounts:
        - name: config-volume
          mountPath: /config
  volumes:
    - name: config-volume
      configMap:
        name: my-config
  restartPolicy: Never
```

ConfigMaps & Secrets

- kubectl apply -f kuard-config.yaml
- kubectl port-forward kuard-config 8080
- point your browser at http://localhost:8080

ConfigMaps & Secrets

- Secrets
  - data that is extra-sensitive - passwords, security tokens, private keys
  - Kubernetes has native support for storing and handling this data with care
  - Secrets enable container images to be created without bundling sensitive data
  - allows containers to remain portable across environments
  - Secrets are exposed to pods via explicit declaration in pod manifests and the Kubernetes API

ConfigMaps & Secrets

- Creating Secrets
  - created using the Kubernetes API or the kubectl command-line tool
  - Secrets hold one or more data elements as a collection of key/value
  - create a secret to store a TLS key and certificate for the kuard application that meets the storage requirements listed above
  - The kuard container image does not bundle a TLS certificate or key
  - This allows the kuard container to remain portable across environments and distributable through public Docker repositories

ConfigMaps & Secrets

- Creating Secrets
  - first step in creating a secret is to obtain the raw data we want to store
  - he TLS key and certificate for the kuard application can be downloaded by running the following commands:
    - curl -o kuard.crt https://storage.googleapis.com/kuar-demo/kuard.crt
    - curl -o kuard.key https://storage.googleapis.com/kuar-demo/kuard.key
  - With the kuard.crt and kuard.key files stored locally, we are ready to create a secret

ConfigMaps & Secrets

- Creating Secrets
  - Create a secret named kuard-tls using the create secret command:
    - kubectl create secret generic kuard-tls --from-file=kuard.crt --from-file=kuard.key
    - kuard-tls secret has been created with two data elements
  - Run the following command to get details:
    - kubectl describe secrets kuard-tls

ConfigMaps & Secrets

- Consuming Secrets
  - Secrets can be consumed using the Kubernetes REST API by applications that know how to call that API directly
  - Instead of accessing secrets through the API server, we can use a secrets volume

ConfigMaps & Secrets

❖ SECRETS VOLUMES

➢ Secret data can be exposed to pods using the secrets volume type

➢ Secrets volumes are managed by the kubelet and are created at pod creation time

➢ Secrets are stored on tmpfs volumes (aka RAM disks) and, as such, are not written to disk on nodes

➢ Each data element of a secret is stored in a separate file under the target mount point specified in the volume mount

➢ The kuard-tls secret contains two data elements: kuard.crt and kuard.key

ConfigMaps & Secrets

- Mounting the kuard-tls secrets volume to /tls results in the following files:
  - /tls/cert.pem
  - /tls/key.pem

kuard-secret.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: kuard-tls
spec:
 containers:
  - name: kuard-tls
    image: gcr.io/kuar-demo/kuard-amd64:1
    imagePullPolicy: Always
    volumeMounts:
    - name: tls-certs
      mountPath: "/tls"
      readOnly: true
 volumes:
  - name: tls-certs
    secret:
      secretName: kuard-tls
```

ConfigMaps & Secrets

- Create the kuard-tls pod using kubectl and observe the log output from the running pod:
  - kubectl apply -f kuard-secret.yaml
- Connect to the pod by running:
  - kubectl port-forward kuard-tls 8443:8443
- Now navigate your browser to https://localhost:8443
- some invalid certificate warnings as this is a self-signed certificate for kuard.example.com
- navigate past this warning, you should see kuard server hosted via HTTPS
- Use the "File system browser" tab to find the certificates on disk

Summary

- ConfigMaps & secrets are a great way to provide dynamic configuration in our application
- They allow us to create a container image (and pod definition) once and reuse it in different contexts
- can include using the exact same image as we move from dev to staging to production
- It can also include using a single image across multiple teams and services
- Separating configuration from application code will make your applications more reliable and reusable

Day 4

- Deployments
  - First Deployment
  - Creating Deployments
  - Managing Deployments
  - Updating Deployments
  - Deployment Strategies
  - Deleting a Deployment
  - Summary

Deployments

- Deployment object exists to manage the release of new versions
- Deployments represent deployed applications in a way that transcends any particular software version of the application

Our First Deployment

- At the beginning we created a Pod by running kubectl run
  - kubectl run nginx --image=nginx:1.7.12
- Under the hood, this was actually creating a Deployment object
  - kubectl get deployments nginx

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx	1	1	1	1	13s

Deployment Internals

- Let's explore how Deployments actually work
- Just as we learned that ReplicaSets manage Pods
- Deployments manage ReplicaSets
- we can see the label selector by looking at the Deployment object:
  - kubectl get deployments nginx -o jsonpath --template {.spec.selector.matchLabels} map[run:nginx]
- From this we can see that the Deployment is managing a ReplicaSet with the labels run=nginx

Deployment Internals

- use this in a label selector query across ReplicaSets to find that specific ReplicaSet:
  - kubectl get replicasets --selector=run=nginx

NAME	DESIRED	CURRENT	READY	AGE
nginx-1128242161	1	1	1	13m

- see the relationship between a Deployment and a ReplicaSet in action
- resize the Deployment using the imperative scale command:
  - kubectl scale deployments nginx --replicas=2

    deployment "nginx" scaled

Deployment Internals

- list that ReplicaSet again, we should see:
  - kubectl get replicasets --selector=run=nginx

NAME	DESIRED	CURRENT	READY	AGE
nginx-1128242161	2	2	2	13m
- Scaling the Deployment has also scaled the ReplicaSet it controls
- try the opposite, scaling the ReplicaSet:
  - kubectl scale replicasets nginx-1128242161 --replicas=1

    replicaset "nginx-1128242161" scaled

Deployment Internals

- Now get that ReplicaSet again:
  - kubectl get replicasets --selector=run=nginx
- That's odd!!
- Despite our scaling the ReplicaSet to one replica, it still has two replicas as its desired state. What's going on?
- Remember, Kubernetes is an online, self-healing system
- The top-level Deployment object is managing this ReplicaSet
- When we adjust the number of replicas to one, it no longer matches the desired state of the Deployment, which has replicas set to 2

Creating Deployments

- download this Deployment into a YAML file:
  - kubectl get deployments nginx --export -o yaml > nginx-deployment.yaml
  - kubectl replace -f nginx-deployment.yaml --save-config

Creating Deployments

❖ If we look in the file, we will see something like this:

```
apiVersion: extensions/v1beta1
kind: Deployment
    metadata:
        annotations:
            deployment.kubernetes.io/revision: "1"
        labels:
            run: nginx
        name: nginx
        namespace: default
```

Creating Deployments

```yaml
spec:
    replicas: 2
    selector:
        matchLabels:
            run: nginx
    strategy:
        rollingUpdate:
            maxSurge: 1
            maxUnavailable: 1
        type: RollingUpdate
    template:
        metadata:
            labels:
                run: nginx
        spec:
            containers:
                - image: nginx:1.7.12
                  imagePullPolicy: Always
            dnsPolicy: ClusterFirst
            restartPolicy: Always
```

Creating Deployments

- NOTE
  - A lot of read-only and default fields were removed in the preceding listing for brevity.
  - We also need to run kubectl replace --save-config.
  - This adds an annotation so that, when applying changes in the future, kubectl will know what the last applied configuration was for smarter merging of configs.
  - If we always use kubectl apply, this step is only required after the first time we create a Deployment using kubectl create -f.
- The Deployment spec has a very similar structure to the ReplicaSet spec.

Creating Deployments

- There is a Pod template, which contains a number of containers that are created for each replica managed by the Deployment.
- In addition to the Pod specification, there is also a strategy object:
  - ```
    ...
     strategy:
      rollingUpdate:
       maxSurge: 1
       maxUnavailable: 1
      type: RollingUpdate
    ...
    ```
- The strategy object dictates the different ways in which a rollout of new software can proceed.
    ```

Creating Deployments

- two different strategies supported by Deployments:
  - Recreate and
  - RollingUpdate
- kubectl describe deployments nginx
- Two of the most important pieces of information in the output are OldReplicaSets and NewReplicaSet
- If a Deployment is in the middle of a rollout, both fields will be set to a value
- If a rollout is complete, OldReplicaSets will be set to <none>
- In addition to the describe command, there is also the kubectl rollout command for deployments

Updating Deployments

- Deployments are declarative objects that describe a deployed application.
- The two most common operations on a Deployment are
  - scaling
  - application updates
- best practice is to manage our Deployments declaratively via the YAML files, and then use those files to update our Deployment.
- scale up a Deployment, edit our YAML file to increase the number of replicas:
  - ```
    ...
    spec:
      replicas: 3
    ...
    ```
    ```

Scaling a Deployment

- update the Deployment using the kubectl apply command:
  - kubectl apply -f nginx-deployment.yaml
  - kubectl get deployments nginx

Updating a Container Image

- other common use case for updating a Deployment is to roll out a new version of the software running in one or more containers
- edit the deployment YAML file, though in this case we are updating the container image, rather than the number of replicas:
  - ...

```
    containers:
    - image: nginx:1.9.10
      imagePullPolicy: Always
...
```

Updating a Container Image

- put an annotation in the template for the Deployment to record some information about the update:
  - ```
    ...
    spec:
      ...
      template:
        metadata:
          annotations:
            kubernetes.io/change-cause: "Update nginx to 1.9.10"
    ...
    ```
    ```

Updating a Container Image

- CAUTION
  - Make sure we add this annotation to the template and not the Deployment itself, since kubectl apply ... uses this field in the Deployment object
  - Also, do not update the change-cause annotation when doing simple scaling operations.
  - A modification of change-cause is a significant change to the template and will trigger a new rollout.
- use kubectl apply to update the Deployment:
  - kubectl apply -f nginx-deployment.yaml

Updating a Container Image

- After we update the Deployment it will trigger a rollout, which we can then monitor via the kubectl rollout command:
  - kubectl rollout status deployments nginx
- Both the old and new ReplicaSets are kept around in case we want to roll back:
  - kubectl get replicasets -o wide

NAME	DESIRED	CURRENT	READY	...	IMAGE(S)	...
nginx-1128242161	0	0	0	...	nginx:1.7.12	...
nginx-1128635377	3	3	3	...	nginx:1.9.10	...

Updating a Container Image

- If we are in the middle of a rollout and want to temporarily pause?
  - kubectl rollout pause deployments nginx
- If, after investigation, we believe the rollout can safely proceed, we can use the resume command to start up where we left off:
  - kubectl rollout resume deployments nginx

Rollout History

- Kubernetes Deployments maintain a history of rollouts, which can be useful both for understanding the previous state of the Deployment and to roll back to a specific version
- see the deployment history by running:
  - kubectl rollout history deployment nginx
- The revision history is given in oldest to newest order
- A unique revision number is incremented for each new rollout
- interested in more details about a particular revision?
- add the --revision flag to view details about that specific revision:
  - kubectl rollout history deployment nginx --revision=2

Rollout History

- Let’s do one more update for this example
- Update the nginx version to 1.10.2 by
  - modifying the container version number and
  - updating the change-cause annotation
  - Apply it with kubectl apply
- Our history should now have three entries:
  - kubectl rollout history deployment nginx

    deployments "nginx"

REVISION	CHANGE-CAUSE
1	<none>
2	Update nginx to 1.9.10
3	Update nginx to 1.10.2

Rollout History

- ❖ Let's say there is an issue with the latest release and we want to roll back while we investigate.
- ❖ simply undo the last rollout:
  - ➢ kubectl rollout undo deployments nginx
- ❖ The undo command works regardless of the stage of the rollout
- ❖ we can undo both partially completed and fully completed rollouts
- ❖ undo of a rollout is actually simply a rollout in reverse (e.g., from v2 to v1, instead of from v1 to v2), and all of the same policies that control the rollout strategy apply to the undo strategy as well

Rollout History

- we can see the Deployment object simply adjusts the desired replica counts in the managed ReplicaSets:
  - kubectl get replicasets -o wide
- CAUTION
  - When using declarative files to control our production systems, we want to, as much as possible, ensure that the checked-in manifests match what is actually running in our cluster

Rollout History

- When we do a kubectl rollout undo we are updating the production state in a way that isn't reflected in our source control.
- An alternate (and perhaps preferred) way to undo a rollout is to revert our YAML file and kubectl apply the previous version.
- In this way our "change tracked configuration" more closely tracks what is really running in our cluster.

❖ Let's look at our deployment history again:

- kubectl rollout history deployment nginx

Rollout History

- Revision 2 is missing!
- It turns out that when we roll back to a previous revision, the Deployment simply reuses the template and renumbers it so that it is the latest revision.
- What was revision 2 before is now reordered into revision 4.
- We previously saw that we can use the kubectl rollout undo command to roll back to a previous version of a deployment.

Rollout History

- Additionally, we can roll back to a specific revision in the history using the --to-revision flag:
  - kubectl rollout undo deployments nginx --to-revision=3
  - kubectl rollout history deployment nginx

    deployments "nginx"

REVISION	CHANGE-CAUSE
1	<none>
4	Update nginx to 1.9.10
5	Update nginx to 1.10.2

Deleting a Deployment

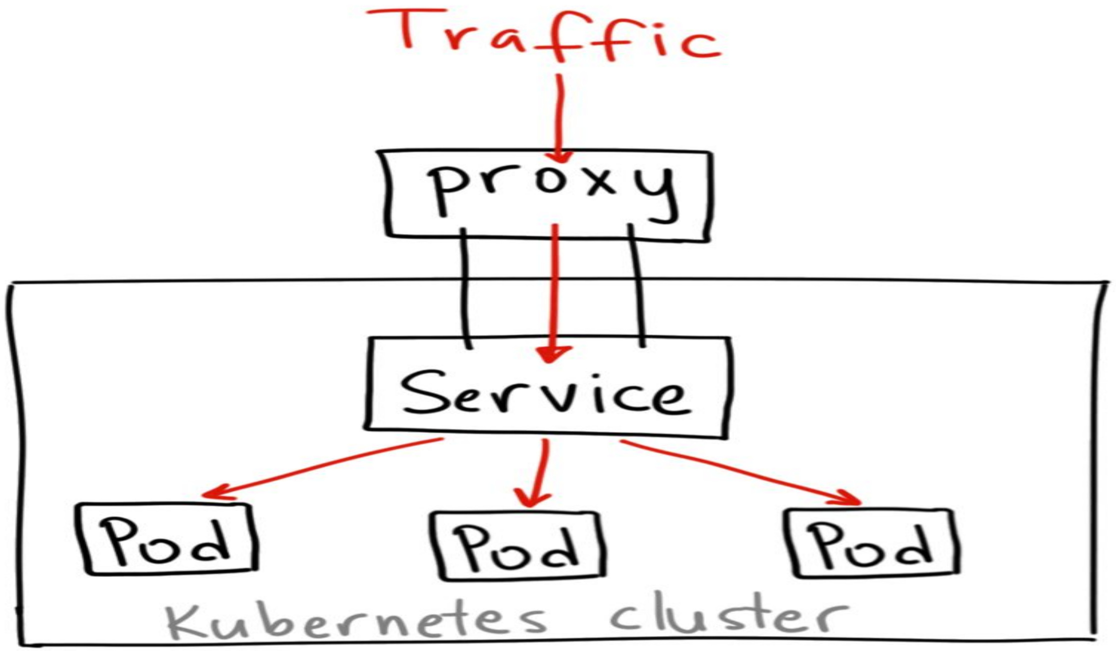
- delete a deployment, we can do it either with the imperative command:
  - kubectl delete deployments nginx
- using the declarative YAML file we created earlier:
  - kubectl delete -f nginx-deployment.yaml
- deleting a Deployment deletes the entire service
- It will delete not just the Deployment, but also any ReplicaSets being managed by the Deployment, as well as any Pods being managed by the ReplicaSets
- As with ReplicaSets, if this is not the desired behavior, we can use the --cascade=false flag to exclusively delete the Deployment object

Ingress

- ClusterIP
  - default Kubernetes service
  - service inside our cluster that other apps inside our cluster can access
  - There is no external access

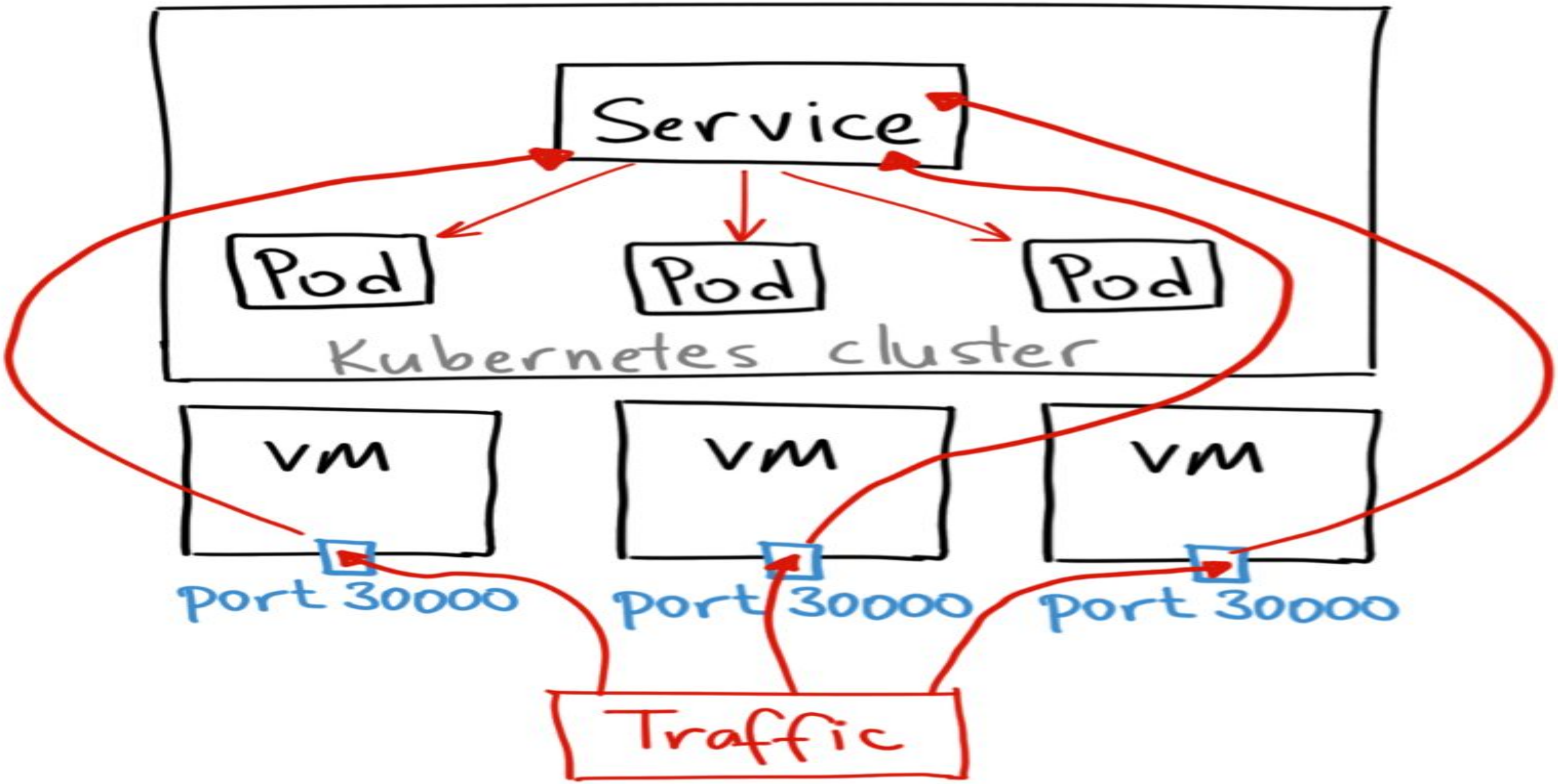
The YAML for a ClusterIP service looks like this:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-internal-service
spec:
  selector:
    app: my-app
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
```



Ingress

- Nodeport

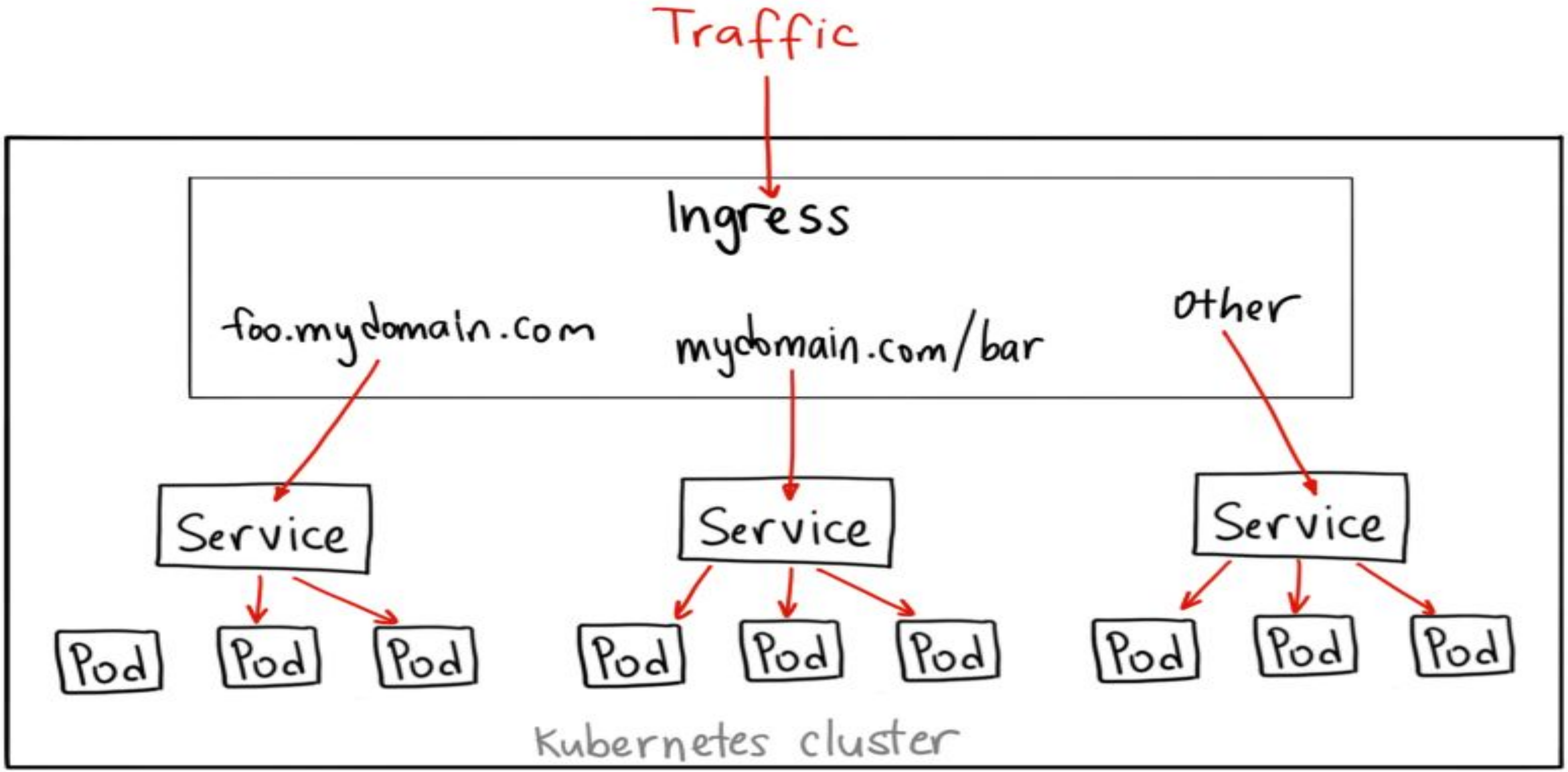


Ingress

- Ingress
  - NOT a type of service
  - sits in front of multiple services and act as a “smart router” or entrypoint into our cluster
  - You can do a lot of different things with an Ingress, and there are many types of Ingress controllers that have different capabilities
  - allows to do both path based and subdomain based routing to backend services

Ingress

❖ Ingress



Ingress

- YAML

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  backend:
    serviceName: other
    servicePort: 8080
  rules:
  - host: foo.mydomain.com
    http:
      paths:
      - backend:
          serviceName: foo
          servicePort: 8080
  - host: mydomain.com
    http:
      paths:
      - path: /bar/*
        backend:
          serviceName: bar
          servicePort: 8080
```

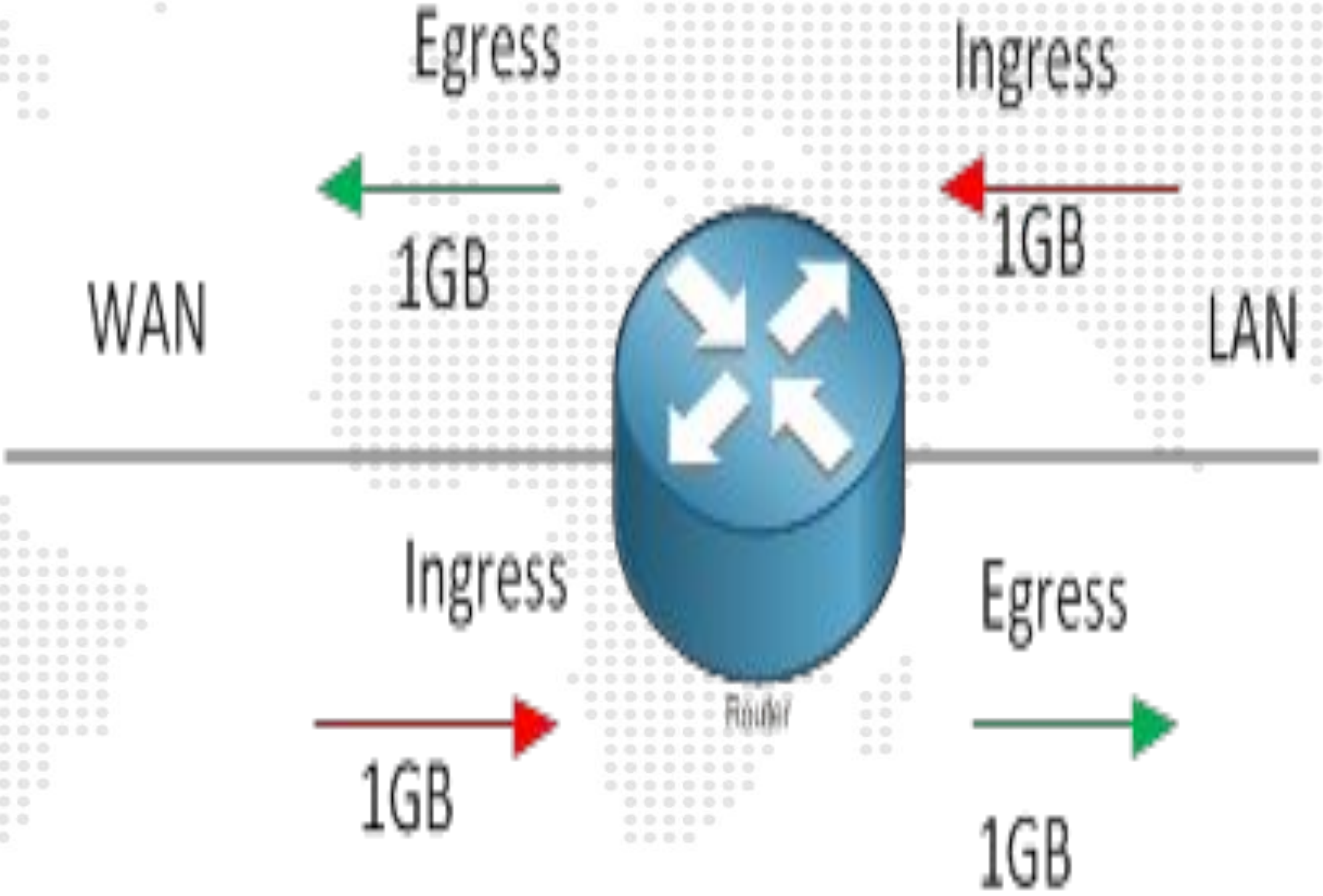
Ingress

- When would you use this?
  - probably the most powerful way to expose your services, but can also be the most complicated
  - many types of Ingress controllers, from the Google Cloud Load Balancer, Nginx, Contour, Istio, and more
  - also plugins for Ingress controllers, like the cert-manager, that can automatically provision SSL certificates for your services
  - most useful if you want to expose multiple services under the same IP address, and these services all use the same L7 protocol (typically HTTP)

Egress

- think for a moment that you are a router
- left hand is the WAN and your right hand is the LAN
- Whenever you say Ingress, it means traffic is towards you, depending on the hand you are looking at
- When you upload data to the internet its going out of your local network so the traffic is egress based on the LAN's perspective but not the router, it will treat that data as ingress since is coming towards it

Egress



Basic Example

- kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/mandatory.yaml
- minikube addons enable ingress
- kubectl get pods --all-namespaces
- kubectl apply -f apple.yaml
- kubectl apply -f banana.yaml
- kubectl create -f ingress.yaml
- curl -kL http://192.168.99.100/apple
- curl -kL http://192.168.99.100/banana
- curl -kL http://192.168.99.100/abc

Cluster

- cat /etc/hosts
  - 192.168.10.40 dockerM
  - 192.168.10.41 docker1
  - 192.168.10.42 docker2
- Docker version validation:
  - yum remove docker-c* -y && yum update -y
  - yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
  - yum install -y yum-utils device-mapper-persistent-data lvm2 -y
  - sudo yum install docker-ce-18.06.1.ce-3.el7 -y

Cluster

- ➢ systemctl enable docker
- ➢ systemctl start docker
- ➢ systemctl start docker && systemctl enable docker

❖ Pre-checks:

- ➢ vim /usr/lib/sysctl.d/00-system.conf

```
# Kernel sysctl configuration file
# For binary values, 0 is disabled, 1 is enabled.  See sysctl(8) and
# sysctl.conf(5) for more details.
# Enable netfilter on bridges.
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-arptables = 1
```

Cluster

- Kubernetes Installation:
  - modprobe br_netfilter
  - echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
  - swapoff -a
- vim /etc/fstab
  - Comment the swap UUID

Cluster

- vim /etc/yum.repos.d/kubernetes.repo

```
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

Cluster

- yum install -y kubelet kubeadm kubectl
- sudo reboot
- modprobe br_netfilter
- echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
- swapoff -a
- systemctl start kubelet && systemctl enable kubelet
- docker info | grep -i cgroup
- sed -i 's/cgroup-driver=systemd/cgroup-driver=cgroupfs/g'

  /etc/systemd/system kubelet.service.d/10-kubeadm.conf
- systemctl daemon-reload
- systemctl restart kubelet

Cluster

- start using our cluster, run the following as a vagrant user:
  - mkdir -p $HOME/.kube
  - sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  - sudo chown $(id -u):$(id -g) $HOME/.kube/config
- Cluster Initialization:
  - kubeadm init --apiserver-advertise-address=192.168.10.40

    --pod-network-cidr=192.168.10.100/24

    Your Kubernetes master has initialized successfully!

Cluster

- join any number of machines by running the following on each node as root:
  - kubeadm join 192.168.10.40:6443 --token 0mtecx.c0tynf5vh5zroc63 --discovery-token-ca-cert-hash sha256:0dfff2cdcd71982cba31ffc3ea0fb29723fa5942bbcdf9eca351e7e1294ec30d

Cluster

- Verify commands:
  - kubectl version
  - kubectl get nodes
  - kubectl get pods --all-namespaces
  - kubectl create deployment nginx --image=nginx
  - kubectl create service nodeport nginx --tcp=80:80
  - kubectl get pods
  - kubectl get svc

Summary

- primary goal of Kubernetes is to make it easy for we to build and deploy reliable distributed systems
- means not just instantiating the application once, but managing the regularly scheduled rollout of new versions of that software service
- Deployments are a critical piece of reliable rollouts and rollout management for our services

- Kubernetes up and Running by Kelsey Hightower, Brendan Burns, Joe Beda
- https://kubernetes.io/docs/reference/glossary/?fundamental=true
- https://www.aquasec.com/wiki/display/containers/Kubernetes+Architecture+101
- https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0
- https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html
- https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727
- https://medium.com/google-cloud/understanding-kubernetes-networking-ingress-1bc341c84078

Thank you for being an awesome Audience!