Kubernetes Foundations

Part Two

Version 1.1.0

About Strigo

- Strigo is a web-based platform that provides the classroom environment for our courses.
- Let’s walk through the features of the platform.
- We’ll also get your lab environment initialized

Introductions

- About your instructor(s)

- Tell us about yourself
  - Would you classify yourself as a
    - developer
    - systems administrator
    - architect
    - It depends on the day/hour
  - What are your goals for learning and adopting Kubernetes?

Agenda

Part One

1. Introduction to Containers
2. Kubernetes Fundamentals
3. Kubernetes Architecture & Troubleshooting
4. Deployment Management
5. Pod & Container Configurations

Agenda

Part Two

6. Kubernetes Networking
7. Resource Organization
8. Storage & Stateful Applications
9. Dynamic Application Configuration
10. Additional Workloads
11. Security

Course Format

- This is a lab-intensive, hands-on course
- Each section will begin with the introduction of a new concept
- Each section contains a lab exercise where you will explore each new concept

Kubernetes Networking

Chapter 06

Agenda

Part Two

6. Kubernetes Networking
7. Resource Organization
8. Storage & Stateful Applications
9. Dynamic Application Configuration
10. Additional Workloads
11. Security

Kubernetes Networking

- Within a Pod
- Pod to Pod
- Services to Pods
- External to Cluster

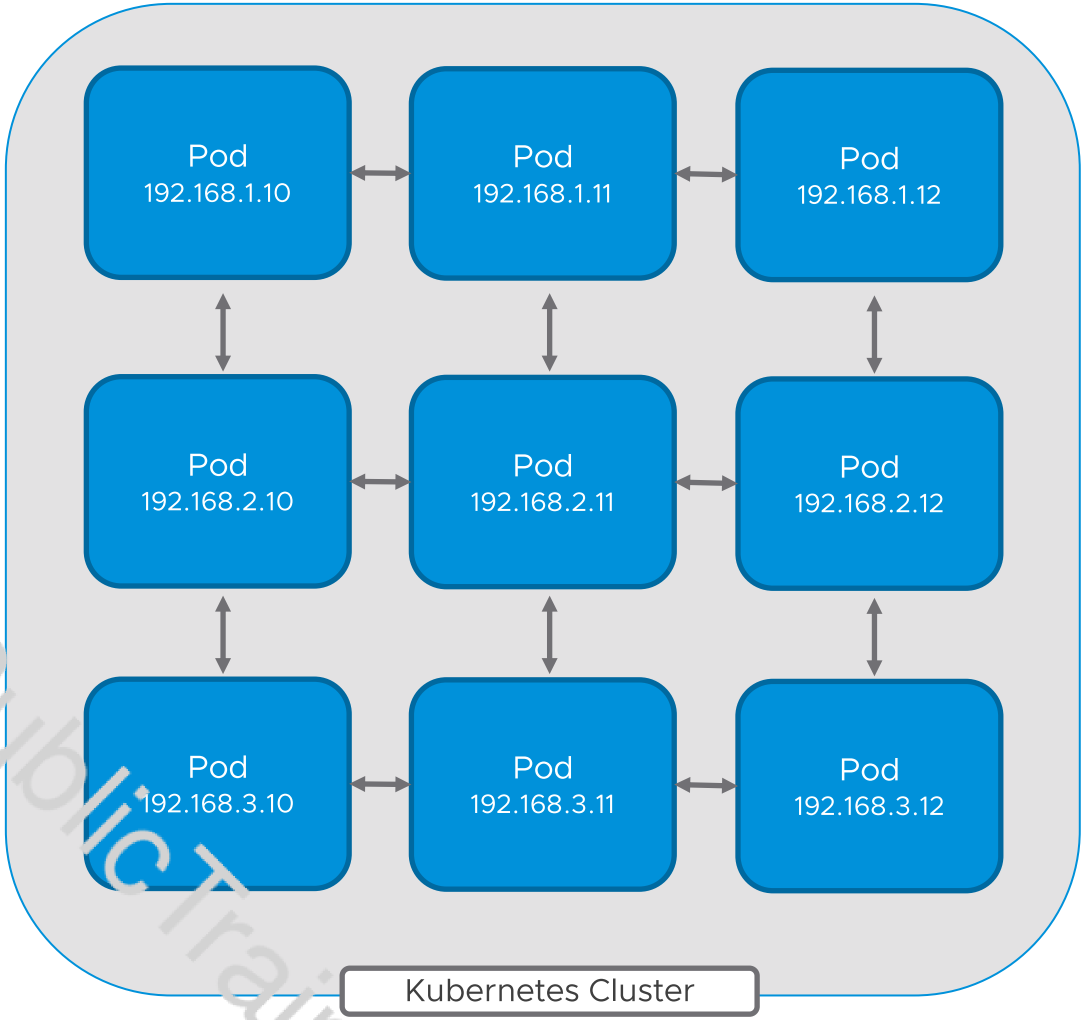
Networking - Within a Pod

- The containers within a pod
  - Can connect to each other using localhost
  - Share an IP address accessible throughout the cluster
  - Share a common port space, beware of conflicts

- These capabilities closely mimic those of multiple processes running on the same virtual machine

Networking - Pod to Pod

- Every pod is assigned an IP address
- This IP address is routable anywhere within the cluster

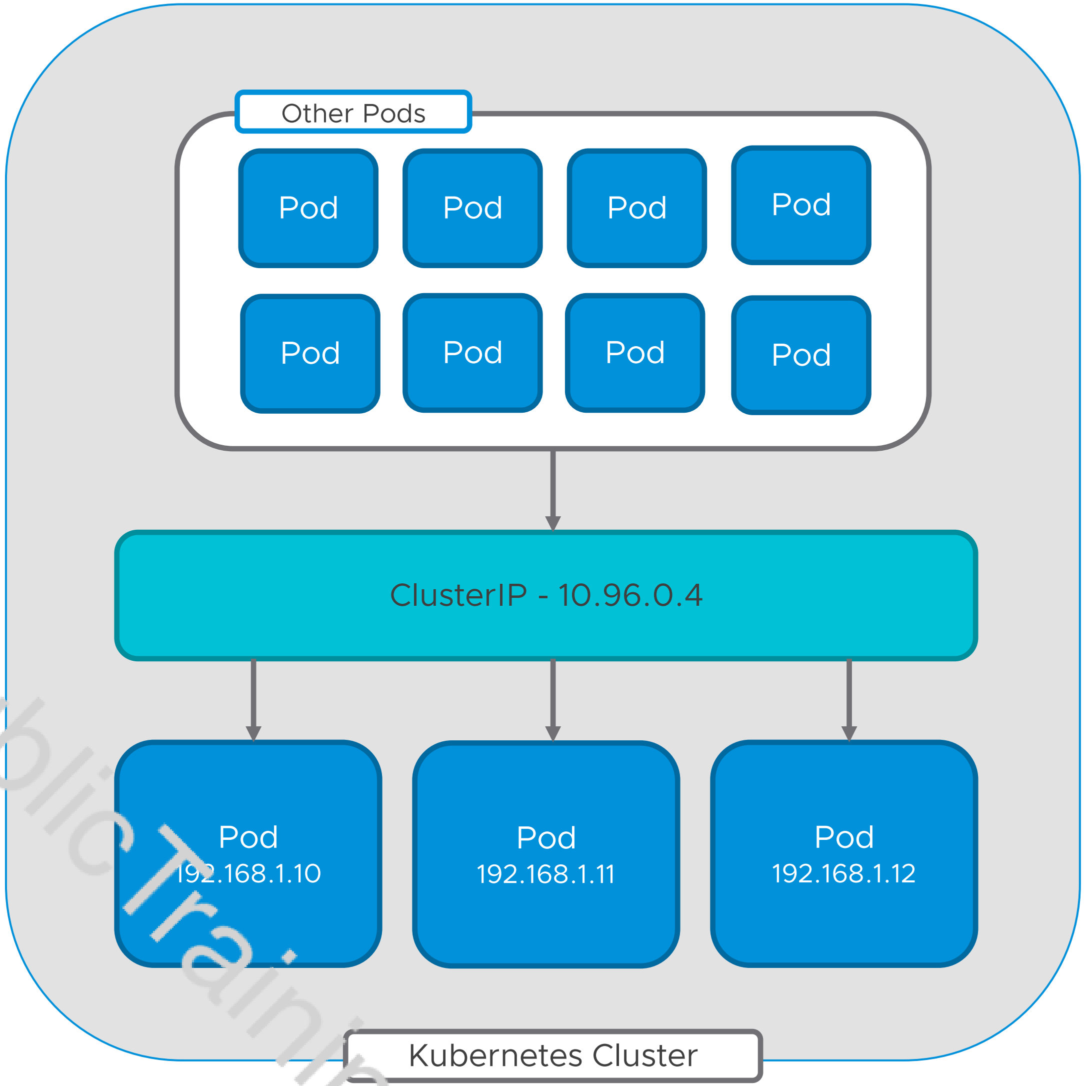


Networking - Services to Pods

- A service is a Kubernetes resource that
  - provides layer-4 load balancing for a group of pods
  - service discovery using the cluster's internal DNS
- Several types of Services are available
  - ClusterIP
  - NodePort
  - LoadBalancer

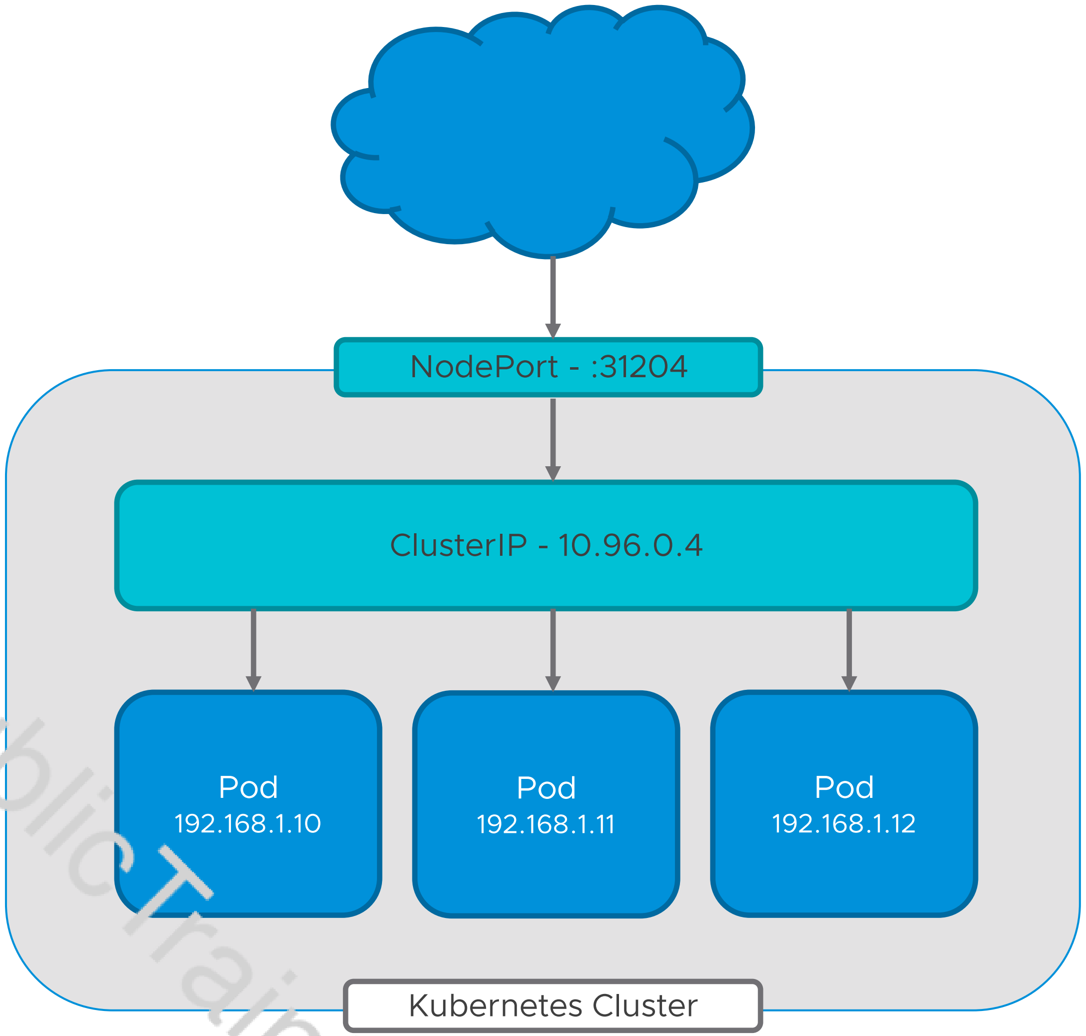
Service Type - ClusterIP

- Used for **internal**-facing services
- Implementation
  - a virtual IP address that load balances requests to a set of backend pods
  - accessible anywhere within the cluster
  - not externally accessible



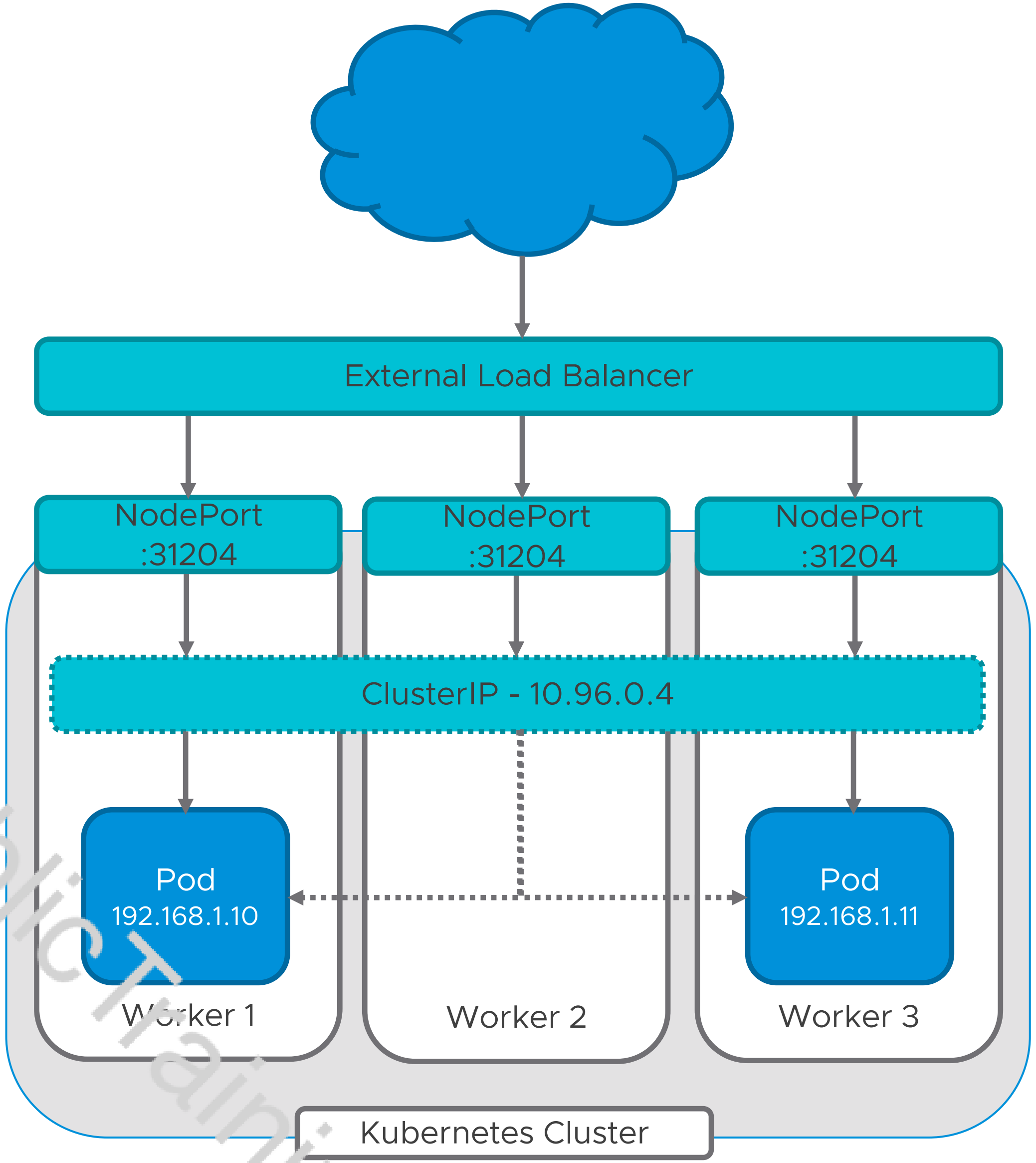
Service Type - NodePort

- Used for **external-facing services**
- Implementation
  - exposes a port on each worker node
  - externally accessible
  - leverages ClusterIP for load balancing to pods



Service Type - LoadBalancer

- Creates and manages an external load balancer
- Implementation
  - leverages NodePort for traffic ingress
  - leverages ClusterIP for load balancing to pods
  - plugins available for different Load Balancer implementations

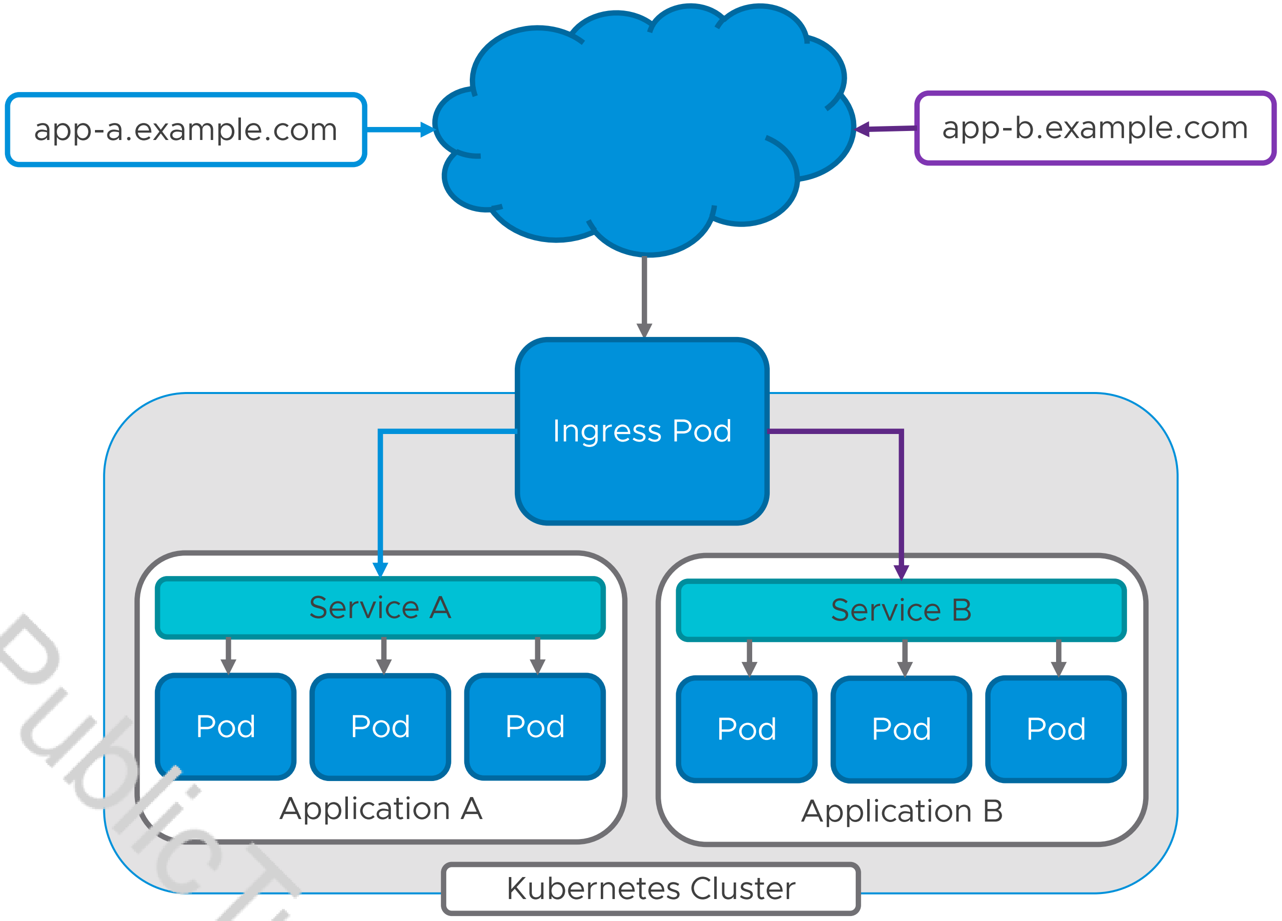


Networking - Ingress Controller to Pods

- An Ingress Controller is a feature which provides
  - layer-7 load balancing for one or more services
  - additional capabilities, depending on implementation
- Many Ingress Controller implementations are available
  - NGINX
  - Contour
  - Traefik
  - Amazon ALB, Google Layer-7 Load Balancer

Ingress Controller

- Used for external-facing layer 7 services
- Implementation
  - uses host header and path evaluation to direct traffic
  - externally accessible
  - configured with Ingress object



Ingress - Example

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-layer7-apps
  annotations:
    nginx.ingress.kubernetes.io/rewrite-
target: /
spec:
```

```
  rules:
  - host: app-a.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: service-app-a
          servicePort: 8080
  - host: app-b.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: service-app-b
          servicePort: 8080
```

Lab 06

Exposing Services

Bootstrap lab to end of lab 05

Deploy an Ingress Controller

Expose Go Web App using an Ingress resource

Resource Organization

Chapter 07

Agenda

Part Two

6. Kubernetes Networking
7. Resource Organization
8. Storage & Stateful Applications
9. Dynamic Application Configuration
10. Additional Workloads
11. Security

Resource Organization - Clusters

- Clusters
  - highest level of isolation but comes with extra management

- Examples
  - environments - prod vs qa vs dev
  - security - compliance requirement vs. no compliance
  - geography - different datacenters

Resource Organization - Namespaces

- Characteristics
  - names of resources must be unique within a namespace
  - scopes dns - <service-name>.<namespace-name>.svc.cluster.local
  - can apply resource and security/access restrictions

- Examples
  - teams - r&d, contractors, etc.
  - systems - email platform, company website, user facing app, etc.

Labels

- Characteristics
  - can exist on basically any resource in Kubernetes
  - nodes have labels too (including built-in ones)
  - keys/values are not enforced
  - not just labels… functionality tied to them (selectors and more)
- Tips
  - keep registration list and report of labels across
  - avoid compound label values:
    - `app: twitter-api` vs `app: twitter / tier: api`
  - be consistent across namespaces and clusters (imagine what labels you would need if everything was in one giant namespace)

kubectl - kubeconfig

- lives at ~/.kube/config

- sections of kubeconfig
  - clusters
  - contexts
  - users

```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://clusterapi
  name: my_prod_cluster
contexts:
- context:
    cluster: my_prod_cluster
    user: admin
    namespace: myapp
  name: admin@kubernetes
current-context: admin@kubernetes
kind: Config
preferences: {}
users:
- name: admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

kubectl - global options

- namespace
  - `--namespace=kube-system`
    - run command for the specified namespace
  - kubectl config set-context <context> --namespace=<namespace>
    - change default namespace for a context

- output
  - -o custom-columns=myName:originalName,myName2:origName2
  - -o [json | yaml] - unfiltered json or yaml
  - -o jsonpath='{.metadata.name}' - filtered json with JSONpath syntax

- sorting
  - --sort-by=.metadata.name

Lab 07

Resource Organization

Exploring namespaces

DNS namespacing

Changing default namespacing

Filtering with kubectl and labels

Storage & Stateful Applications

Chapter 08

Agenda

Part Two

6. Kubernetes Networking
7. Resource Organization
8. Storage & Stateful Applications
9. Dynamic Application Configuration
10. Additional Workloads
11. Security

Storage

- Volumes
  - exposed at Pod level and backed different ways
  - `emptyDir` - ephemeral scratch directory that lives for the life of pod
- Persistent Volumes
  - abstraction away from the storage provider (AWS EBS, GCE PD)
  - have a lifecycle independent of any individual pod that uses it
  - `hostPath` – requires node path knowledge; typically only for Ops use
- Persistent Volume Claims
  - request for storage with specific details (size, access modes, etc.)

Persistent Volumes

- By default, containers write to ephemeral storage
- As a result, when a pod is terminated, all data written by it's containers is lost
- We can attach Persistent Volumes to pods which persist any data written to them

Persistent Volumes

- Persistent volumes can be provisioned either statically or dynamically.
  - Static: A pre-provisioned pool of volumes such as iSCSI or Fiber Channel disk from a SAN
  - Dynamic: Volumes are created on demand by calling a storage provider’s API such as Amazon EBS
- Persistent Volumes have a lifecycle independent of the pods that use them.

Persistent Volume Claims

- Created by users to request a persistent volume
- Users can request various properties such as capacity and access modes (e.g., can be mounted once read/write or many times read-only)

Using Persistent Volumes

PersistentVolumeClaim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mysql-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
```

PVC in a Pod

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: my-mysql
      image: mysql:5.6
      volumeMounts:
      - mountPath: "/var/lib/mysql"
        name: mysql-pd
  volumes:
    - name: mysql-pd
      persistentVolumeClaim:
        claimName: mysql-pvc
```

StatefulSets

- stable, unique pod indentifiers and DNS names
  - ${statefulset_name}-${ordinal}.${service_name}.${namespace}
- stable, persistent storage
  - one PV per VolumeClaimTemplate
  - sticky to pod for as long as the pod is declared in the API
- Controlled deployment order
  - pods created in asc order; deleted in desc order
  - before scaling all pods must be Running or Ready

StatefulSet - Example

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: my-mysql
  serviceName: my-mysql
replicas: 1
  template:
    metadata:
      labels:
        app: my-mysql
```

```yaml
spec:
  containers:
  - name: mysql
    image: mysql:5.6
    volumeMounts:
    - name: mysql-pd
      mountPath: /var/lib/mysql
  volumeClaimTemplates:   # optional
   - metadata:
       name: mysql-pd
     spec:
       accessModes: [ "ReadWriteOnce" ]
       storageClassName: "my-storage-class"
       resources:
         requests:
           storage: 8Gi
```

Lab 08

Stateful Applications

Convert gowebapp-mysql Deployment to a StatefulSet

Simulate a pod failure

Verify data persistence

Dynamic Application Configuration

Chapter 09

Agenda

Part Two

6. Kubernetes Networking
7. Resource Organization
8. Storage & Stateful Applications
9. Dynamic Application Configuration
10. Additional Workloads
11. Security

Image Configurability

- All runtime configuration should be able to be injected and overridden
  - defaults inside a container image are fine
  - enables configuration to be supplied by Kubernetes and the environment

- Author can choose two approaches
  - provide defaults and startup if no configuration is provided
  - require configuration and fail to startup if none is provided

ConfigMaps

- Created from YAML or `kubectl create`
  - application build can produce ConfigMaps for each environment
  - namespace specific

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myapp-config-qa
  namespace: myapp-qa
data:
  server.properties: \
    MYAPP_MAX_THREADS = 25
    MYAPP_TLS_ENABLED = false
  job.properties: \
    MYJOB_NUM_THREADS = 5
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myapp-config-prod
  namespace: myapp-prod
data:
  server.properties: \
    MYAPP_MAX_THREADS = 100
    MYAPP_TLS_ENABLED = true
  job.properties: \
    MYJOB_NUM_THREADS = 10
```

Using ConfigMaps - As Environment Variables

- Load All Keys

```
image: myapp:v1.0.0
envFrom:
- configMapRef:
    name: myapp-config-prod
```

- Load Individual Keys

```
image: myapp:v1.0.0
env:
  - name: MYAPP_MAX_THREADS
    valueFrom:
      configMapKeyRef:
        name: myapp-config-prod
        key: MYAPP_MAX_THREADS
```

Using ConfigMaps - As Mounted Volumes

Load All Keys

```
containers:
  - name:
    image: myapp:v1.0.0
    volumeMounts:
    - name: myconfigvolume
      mountPath: /app/config
volumes:
  - name: myconfigvolume
    configMap:
      name: myapp-config-prod
```

Load Individual Key

```
containers:
  - name:
    image: myapp:v1.0.0
    volumeMounts:
    - name: myconfigvolume
      mountPath: /app/config
volumes:
  - name: myconfigvolume
    configMap:
      name: myapp-config-prod
      items:
      - key: MYAPP_MAX_THREADS
        path: maxWidgets
```

Using ConfigMaps - Variables vs Volumes

- Environment Variables
  - not reliant on filesystem nuances
  - easier when running containers outside of Kubernetes
  - changes require restart

- Mounted Volume
  - updates to ConfigMaps are reflected on the mounted volume

Secrets

- Resource that stores sensitive data such as a password, a token, or a key
  - Must be Base64 encoded
  - Like ConfigMaps, secrets can be exposed to a pod via environment variables and mounted volumes.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Lab 09

Modify Docker Image

Update the gowebapp frontend image

Test application locally

Secrets and ConfigMaps

Create a secret for the MySQL password

Create a ConfigMap for gowebapp's configuration

Update the gowebapp and mysql deployments

Additional Workloads

Chapter 10

Agenda

Part Two

6. Kubernetes Networking
7. Resource Organization
8. Storage & Stateful Applications
9. Dynamic Application Configuration
10. Additional Workloads
11. Security

Jobs

- Characteristics
  - run to completion
  - container/pod based
  - ensured to run to completion; retry forever by default

- Post job completion
  - Pod and Job resources will remain so can still view logs, output, etc.
  - up to the administrator to implement/determine cleanup rules

Jobs

An example Job that computes π to 2000 places

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
```

Job Types

- Non-parallel Jobs
  - "normal" job with one pod. Job completes when Pod terminates successfully
- Parallel Jobs - fixed completion count
  - complete when there is one successful pod for each value in the range 1 to .spec.completions
  - example: process 1mil sequential records across 10 parallel jobs
- Parallel Jobs - work queue
  - when any pod terminates with success, no new pods are created
  - coordination is down with external service and all pods exist at same time

CronJobs

- Types
  - scheduled once at specified time
  - repeated at specified time

- Important Notes
  - new Job resource objects created for each run
  - by default 3 successful jobs are and 1 failed jobs are retained
  - jobs should be idempotent

CronJobs

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

DaemonSets

- A DaemonSet ensures that all Nodes run a copy of a Pod.
  - As nodes are added to the cluster, Pods are added to them
  - As nodes are removed from the cluster, those Pods are deleted
  - Often used for cluster administration functions

- Common use cases
  - Cluster-wide logging agents
  - Cluster-wide monitoring agents

Lab 10

Additional Workloads

Create a Job

Create a CronJob

Create a ParallelJob

Security

Chapter 11

Agenda

Part Two

6. Kubernetes Networking
7. Resource Organization
8. Storage & Stateful Applications
9. Dynamic Application Configuration
10. Additional Workloads
11. Security

Network Policy

Network Policy

- Specification of how groups of pods are allowed to communicate with:
  - each other
  - other network endpoints

- Use labels to select pods and define rules which specify what traffic is allowed to the selected pods

Isolated and Non-isolated Pods

- Pods are non-isolated (default)
  - Accept traffic from any source
- Pods become isolated by:
  - Defining a NetworkPolicy that selects them in a Namespace
  - Pod will reject any connections that are not explicitly allowed by a NetworkPolicy
  - Other Pods in the Namespace that are not selected by any NetworkPolicy will continue to accept all traffic

Network Policy Example – Inbound

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      tier: backend
  policyTypes:
  - Ingress
```

```yaml
  ingress:
  # pods in gowebapp namespace OR are labeled gowebapp
  - from:
    - namespaceSelector:
        matchLabels:
          project: gowebapp
    - podSelector:
        matchLabels:
          app: gowebapp
    ports:
    - protocol: TCP
      port: 3306
```

Network Policy Example – Inbound Improved

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      tier: backend
  policyTypes:
  - Ingress
```

```yaml
  ingress:
    # pods in gowebapp namespace AND are in tier frontend
    - from:
      - namespaceSelector:
          matchLabels:
            project: gowebapp
    - from:
      - podSelector:
          matchLabels:
            tier: frontend
      ports:
      - protocol: TCP
        port: 3306
```

Network Policy Example – Inbound and Outbound

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      tier: backend
  policyTypes:
  - Ingress
  - Egress
```

```yaml
  ingress: # pods in gowebapp namespace AND are tier frontend
  - from:
    - namespaceSelector:
        matchLabels:
          project: gowebapp
  - from:
    - podSelector:
        matchLabels:
          tier: frontend
    ports:
    - protocol: TCP
      port: 3306
  egress: # allow the DB to connect to LDAP for auth
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 389
```

Requesting Escalated Privileges for Pods and Containers

SecurityContext

- PodSpec section for defining container privileges
  - if defined at pod level, then defines defaults for all containers
  - can be overridden and specified per container as well
  - cluster administrators can enforce restrictions with policies

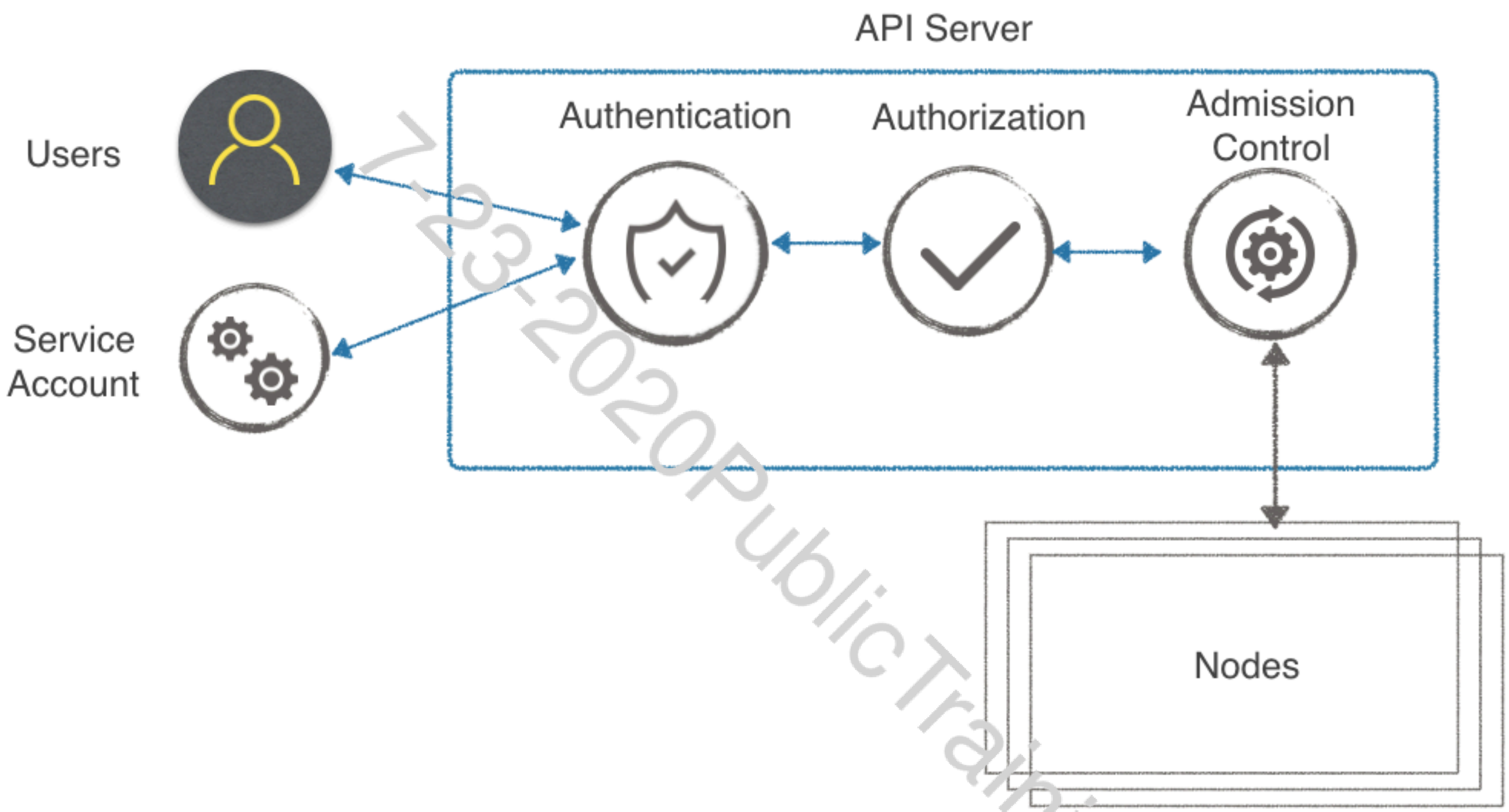
- Example settings
  - runAsUser / runAsGroup
  - SELinux
  - Linux Capabilities
  - Host file paths allowed to be mounted and accessed

SecurityContext - Example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: myapp
    image: myapp:v1
    securityContext:
      allowPrivilegeEscalation: false
  - name: sidecar
    image: mysidecar:v1
    securityContext:
      runAsUser: 2000
```

Controlling Access

API Access Control



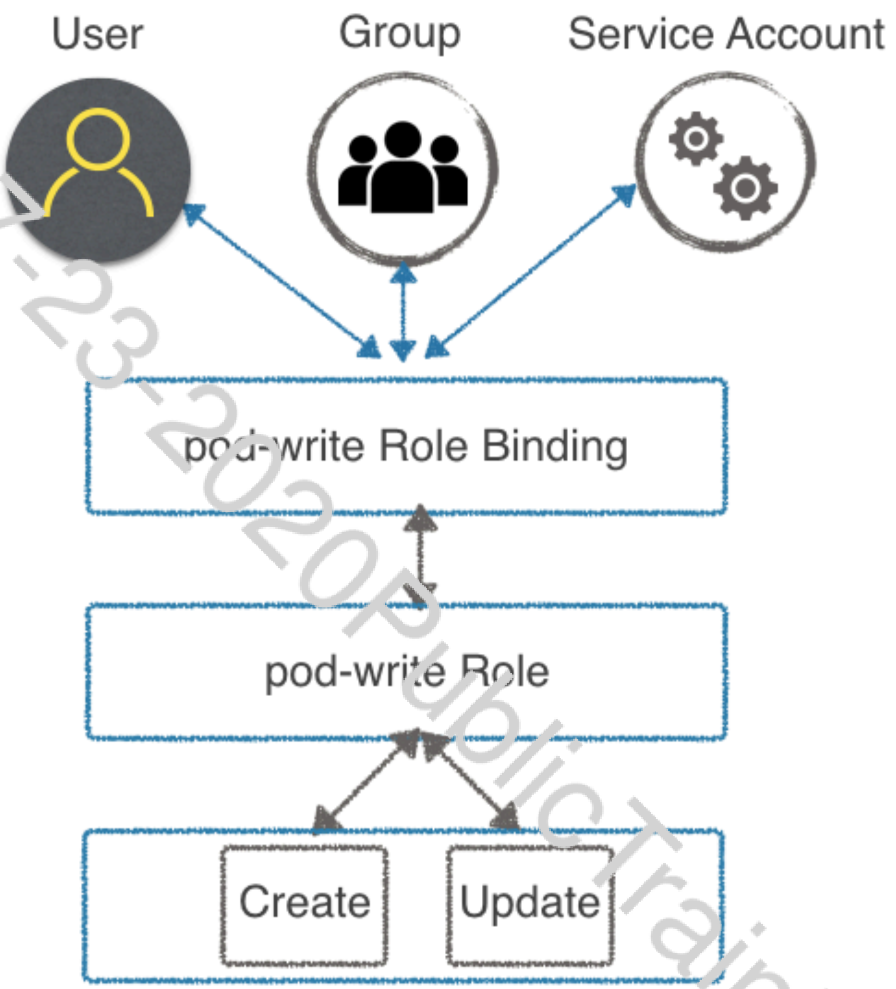
Authentication Methods

- Client Certificates
  - common usage: Cluster Components
- Tokens
  - common usage: Service Accounts
- External Authentication
  - common usage: Users

Authentication - Service Accounts

- Service Account Tokens
  - generated automatically when a ServiceAccount object is created
  - mounted inside pods /var/run/secrets/kubernetes.io/serviceaccount
  - spec.serviceAccountName - overrides default service account

Role-Based Access Control (RBAC)



Role

- Characteristics
  - collection of permissions (rules)
  - standalone and must be bound to a subject
  - namespace specific

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: my-app-secret-reader
rules:
- apiGroups: [""]
  resources: ["secret"]
  verbs: ["get"]
```

ClusterRole

- Characteristics
  - same as Role, except is global to the cluster
  - reusable across entire cluster

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```


RoleBinding

- Characteristics
  - connects a one or more subjects to a Role/RoleBinding

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: my-app-read-secret
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-app
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: my-app-secret-reader
  apiGroup: rbac.authorization.k8s.io
```

ClusterRoleBinding

- Characteristics
  - Same as RoleBinding, except is global to the cluster

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Built-in Roles

Default ClusterRole	Description
cluster-admin	Allows super-user access to perform any action on any resource. When used in a **ClusterRoleBinding**, it gives full control over every resource in the cluster and in all namespaces. When used in a **RoleBinding**, it gives full control over every resource in the rolebinding's namespace, including the namespace itself.
admin	Allows admin access, intended to be granted within a namespace using a **RoleBinding**. If used in a **RoleBinding**, allows read/write access to most resources in a namespace, including the ability to create roles and rolebindings within the namespace. It does not allow write access to resource quota or to the namespace itself.
edit	Allows read/write access to most objects in a namespace. It does not allow viewing or modifying roles or rolebindings.
view	Allows read-only access to see most objects in a namespace. It does not allow viewing roles or rolebindings. It does not allow viewing secrets, since those are escalating.

RBAC Example

RBAC Example

- Jack should have read-only access to the cluster

RBAC Example

- Jack should have read-only access to the cluster

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: my-global-access
subjects:
- kind: User
  name: Jack
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: view
  apiGroup: rbac.authorization.k8s.io
```

RBAC Example

- Jack should also have edit access to "foo" namespace

RBAC Example

- Jack should also have edit access to "foo" namespace

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: my-team-access
  namespace: foo
subjects:
- kind: User
  name: Jack
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
```

Lab 11

Security

Apply network policies

Create a service account

Apply permissions to a service account

Conclusion

Thank You