Microsoft

Module 4: Core Kubernetes Concepts

Agenda

- Introduction to Kubernetes
- Kubernetes Clusters
- Pods, Replica Sets and Deployments
- Deployment Strategies
- Networking Services
- Config Maps and Secrets
- Namespaces
- KubeConfig

1

Introduction to Kubernetes

Benefit of Using Containers

- Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
- Cloud and OS distribution portability: Runs on-premises, on major public clouds and anywhere else.
- Environmental consistency across development, testing, and production: ***Runs the same on a laptop as it does in the cloud***.
- Loosely coupled, distributed, liberated micro-services: applications are broken into ***smaller, independent pieces and can be deployed and managed dynamically*** – not a monolithic stack running on one big single-purpose machine.

Containers in Production

- In a production environment, you need a way to manage containers and ensure that there is no downtime.
- For example, if a container goes down, another container instance needs to start in its place. Wouldn't it be great if this happened automatically?
- What about increased load on the container? It would be great to create multiple instances of a container and then use a load balancer to split traffic across those instances.
- This type of behavior is available natively when using a Container ***Orchestrator***.
- While there are several orchestrators available to choose from, the industry leader is ***Kubernetes***.

What is Kubernetes?

- Kubernetes is a portable, extensible, open-source **platform for managing containerized workloads and services**, that facilitates both declarative configuration and automation.
- The name *Kubernetes* originates from Greek, meaning helmsman or pilot.
- **K8s** as an abbreviation results from counting the eight letters between the "K" and the "s".
- Google open-sourced the Kubernetes project in 2014. Donated to Cloud Native Computing Foundation (CNCF) in 2015
- Microsoft's Brendan Burns was a co-creator!

What Does Kubernetes Do?

Management/automation across the multiple hosts inside a cluster

Scheduling
Provision container instances

Affinity/anti-affinity
Provisioning nearby or far apart: Availability/performance

Health monitoring
Detect and fix failures

Failover
Re-provision instances to healthy machines

Scaling
Add/remove instances to meet demand

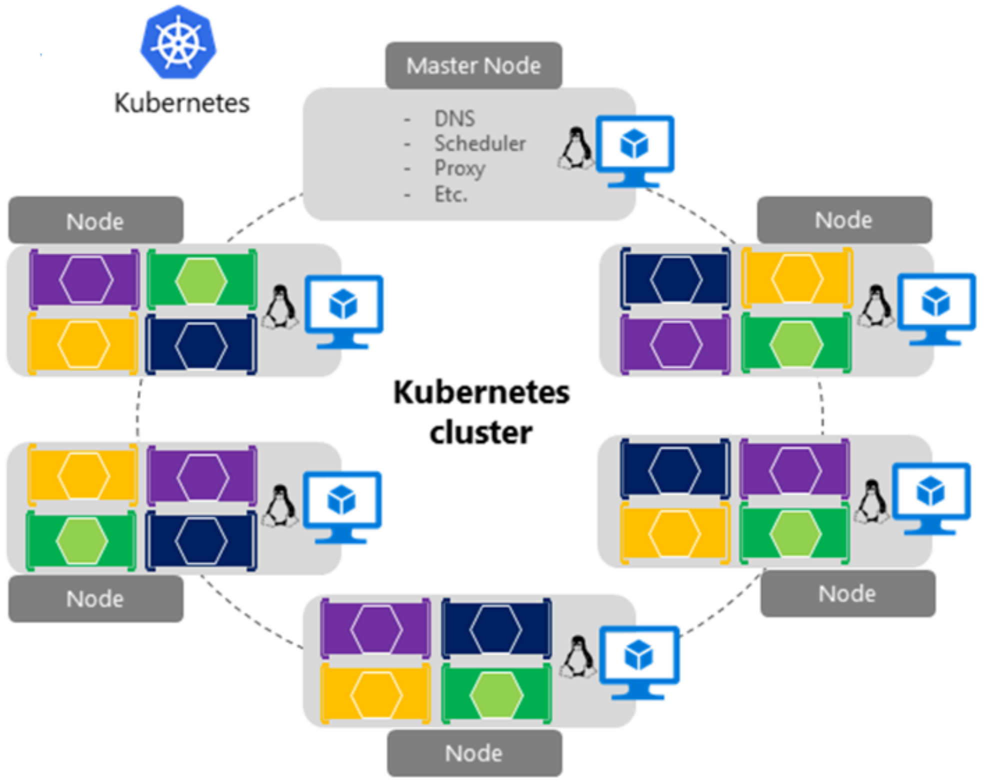
Networking
Networking overlay for container communication

Service discovery
Enable containers to locate each other

Coordinated app upgrades
Avoid downtime and automatically rollback

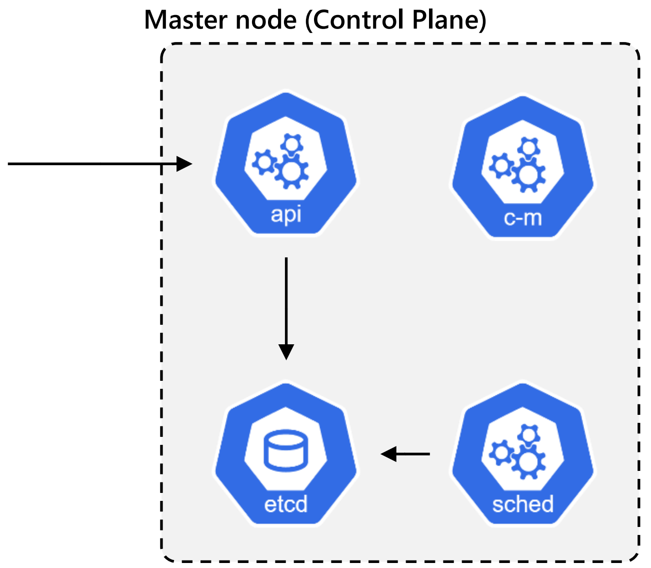
What is a Kubernetes Cluster?

- A ***Kubernetes cluster*** consists of a set of machines (physical or virtual), called ***nodes***.
- A cluster forms highly-available environment.
- The ***master node*** (aka control plane) manages the worker nodes and the cluster.
- The ***worker node(s)*** host the containerized workloads and networking that form the components of the application.
- Worker nodes can be ***heterogeneous*** – some are small, some large, some with GPUs, some running Windows, etc.



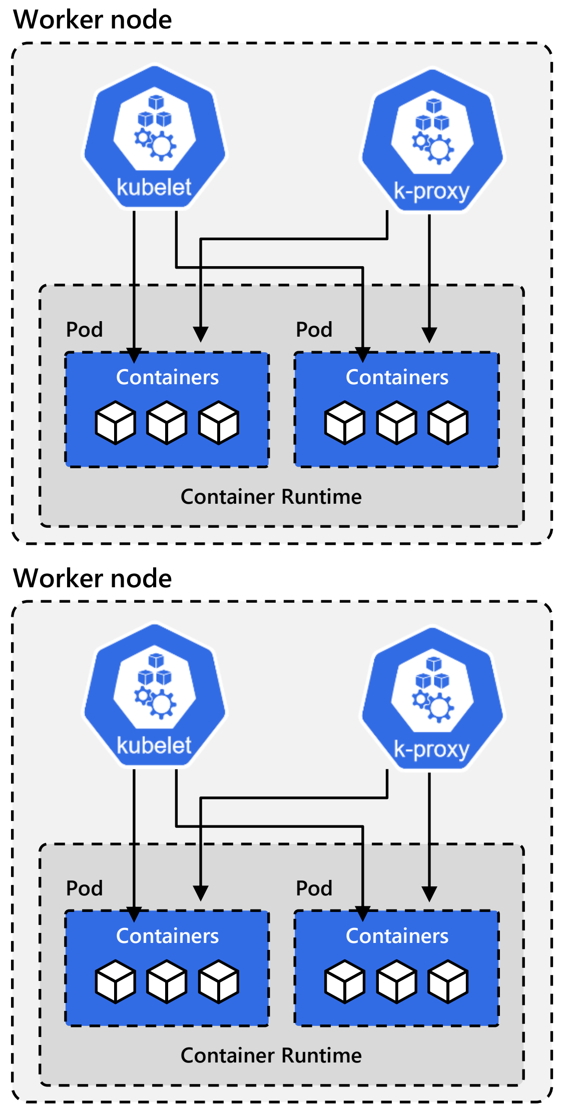
Kubernetes Cluster – Control Plane Components

- **API Server** - Exposes the Kubernetes API outside the cluster.
- **etcd** - Consistent and highly-available key value store used to store for all cluster data.
- **Scheduler** - Watches for newly created Pods with no assigned node and selects a node for them to run on.
- **Controller Manager** - Manages controller processes.



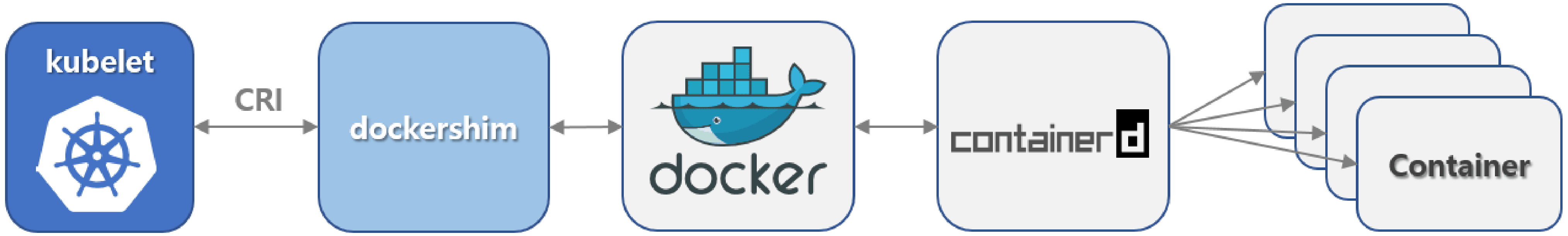
Kubernetes Cluster – Worker Node Components

- **Kubelet** - An agent that runs on each node in the cluster. It manages everything that happens on the node.
- **Kube-proxy** - A network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.
- **Container runtime** - The engine responsible for running containers.
  - Kubernetes clusters have a choice of *container runtimes* to use:
    - **Docker**
    - **containerd**



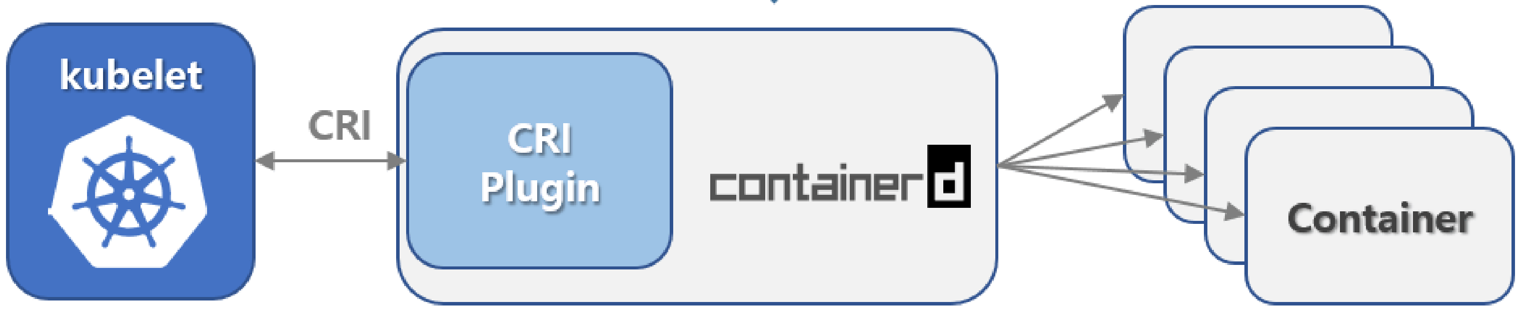
Kubernetes is “Deprecating” Docker

- The Kubernetes team announced that Docker is being “Deprecated” as of 1.22.
- Is this a problem? Should this influence your use of Docker containers?
- Docker was created before Kubernetes. It has its own method of interacting with containers. It uses an *internal* component called ***containerd*** as its *Container Runtime*.
- For Kubernetes to be compatible with Docker container, it used to use a module called ***dockershim*** to connect to ***Docker***, and have Docker run its containers using ***containerd***.



Kubernetes Going Forward

- Kubernetes published a plugin interface called ***Container Runtime Interface (CRI)***, which enables ***kubelet*** to use a wide variety of container runtimes.
- Docker released a *stand-alone* version of the ***containerd*** runtime engine.
- Kubernetes now uses a ***CRI Plugin*** to access ***containerd*** without Docker.



- This change makes Kubernetes run more efficiently.
- **This has no effect on the format, size or performance of Docker containers running in Kubernetes!**
- Feel free to **ignore all the FUD** (*Fear, Uncertainty and Doubt*) around all the "*Kubernetes is deprecating Docker*" chatter online.

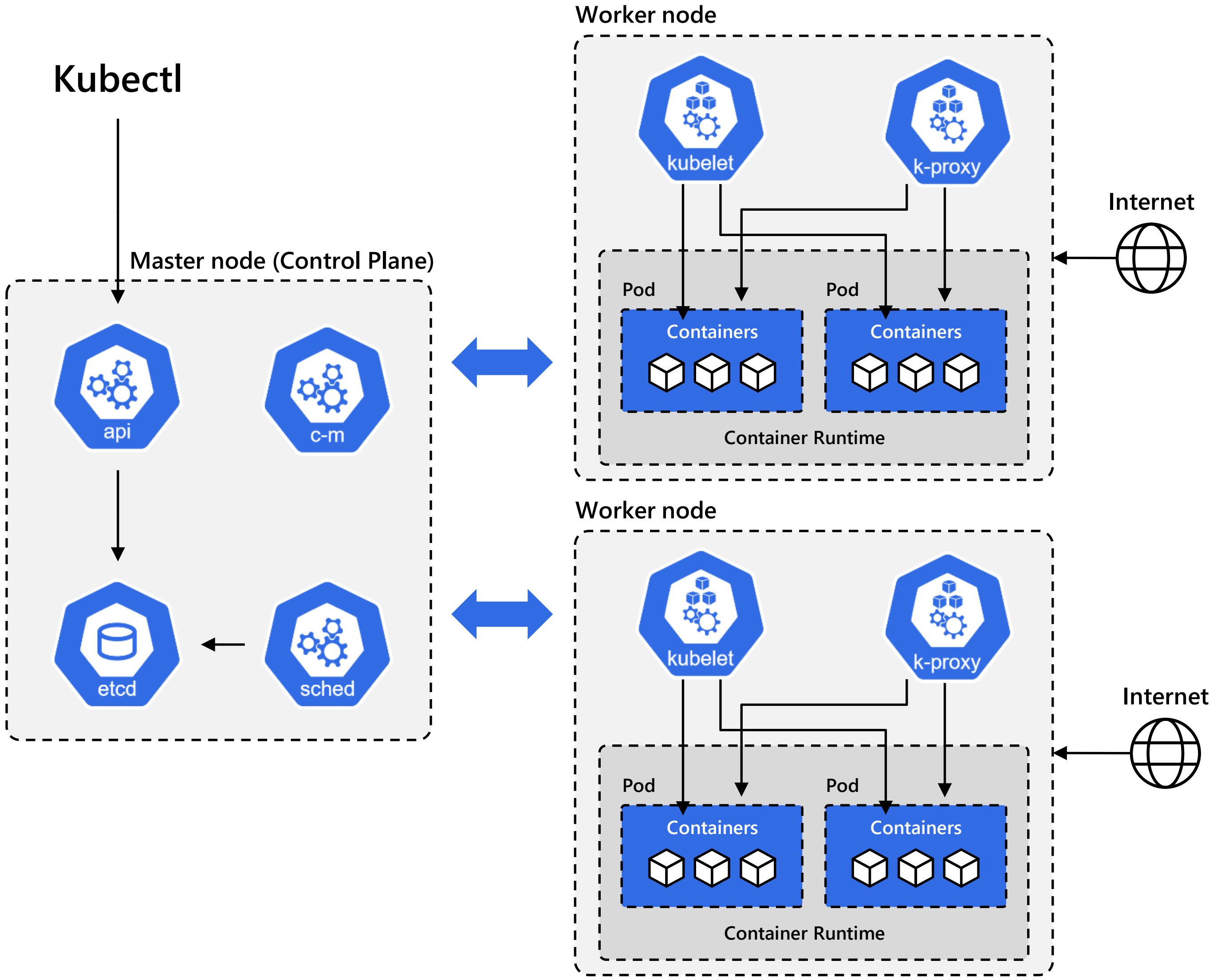
Managing a Kubernetes Cluster

Since Kubernetes is a simple REST API application, you can manage a cluster by making REST calls to the API Server.

Example HTTP Request:

```
GET /api/v1/namespaces/default/pods/{name}
```

However, it's much easier to use the official Kubernetes client command-line utility (**kubectl**), instead.

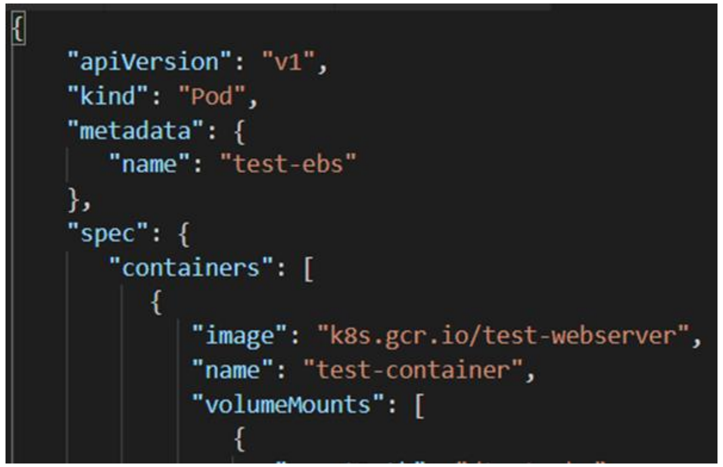


Declarative Configuration

- An ***imperative configuration*** explicitly instructs a system on the steps to take to achieve a desired outcome (like using Docker commands):
  - Connect to container registry
  - Pull desired image
  - Create container
  - Start container
- A ***declarative configuration*** specifies a the final, or <u>*desired state*</u> of an object, and lets the system determine what steps to take to achieve that state.
- The Kubernetes control plane continually and actively manages every object's <u>**actual state to match the desired state**</u> you supplied.

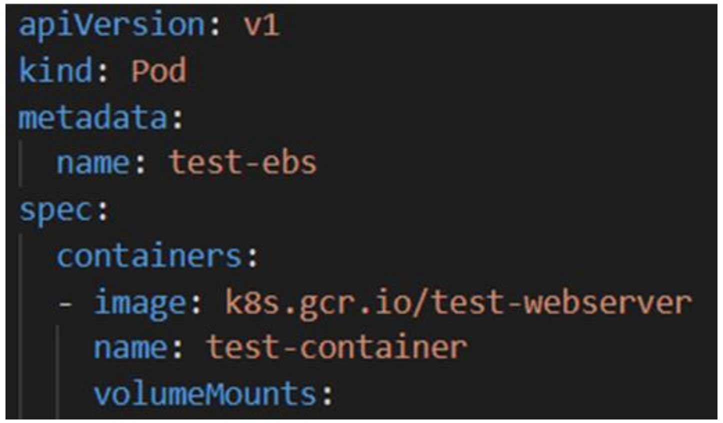
Declaring Kubernetes Object In YAML

- When working with REST API applications like Kubernetes, you expect to exchange JSON data.
- However, when using **kubectl**, you provide a desired state configuration using the **YAML** markup language (**Y**et **A**nother **M**arkup **L**anguage) instead.
- **YAML** uses indentation instead nested curly brackets ({}) to define structure and hierarchy.
- **kubectl** converts your YAML to JSON when making communicating with the Kubernetes API Server.



```
{
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
        "name": "test-ebs"
    },
    "spec": {
        "containers": [
            {
                "image": "k8s.gcr.io/test-webserver",
                "name": "test-container",
                "volumeMounts": [
                    {
```

JSON

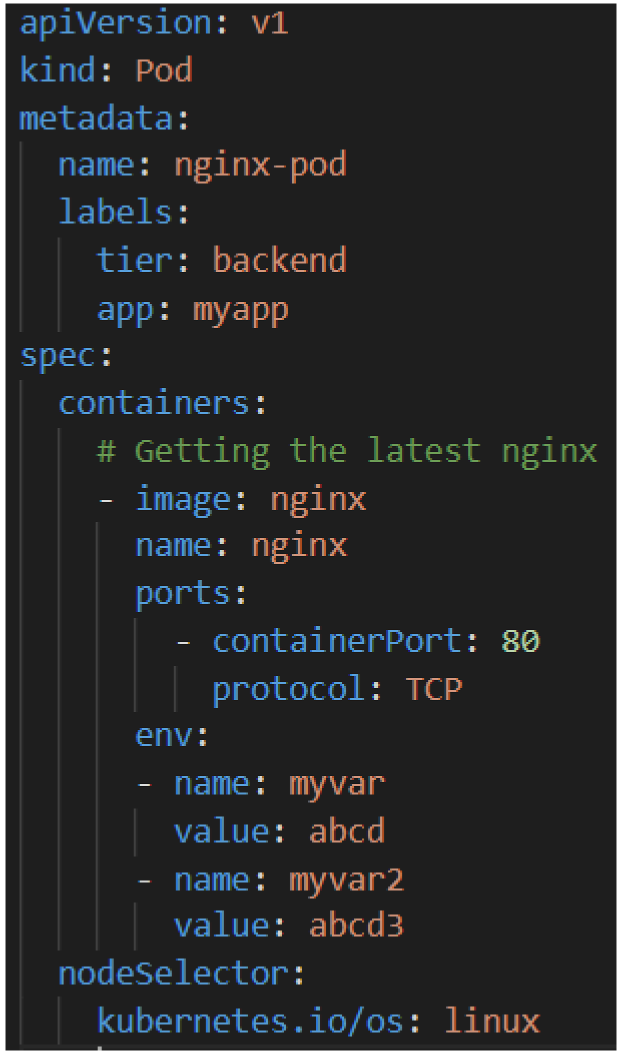


```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
```

YAML

YAML – Yet Another Markup Language

- Whitespace indentation is used for denoting structure.
- Tab characters are *not allowed* as part of that indentation.
- Comments begin with the number sign (**#**), can start anywhere on a line and continue until the end of the line.
- List members are denoted by a leading hyphen (**-**) with one member per line.
- An associative array entry is represented using colon space in the form **key: value** with one entry per line.
- Strings are ordinarily unquoted but may be enclosed in double-quotes ("), or single-quotes (‘).
- Multiple documents with single streams are separated with 3 hyphens (**---**).

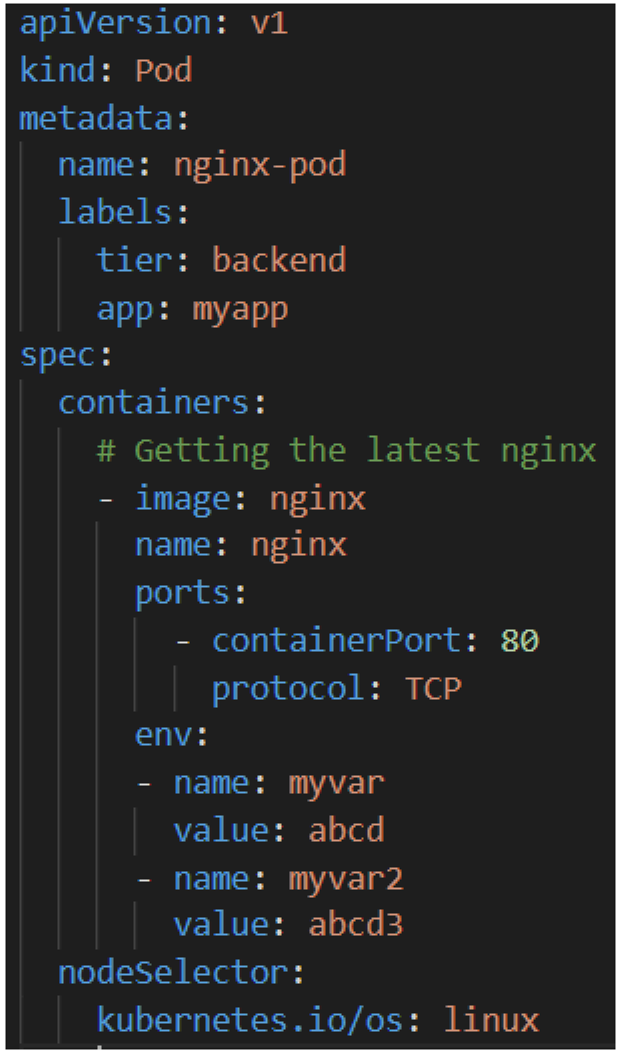


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    tier: backend
    app: myapp
spec:
  containers:
    # Getting the latest nginx
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
          protocol: TCP
      env:
      - name: myvar
        value: abcd
      - name: myvar2
        value: abcd3
  nodeSelector:
    kubernetes.io/os: linux
```

Kubernetes Object Definitions

The Kubernetes API requires the following fields be specified in all YAML files (***a.k.m.s.***):

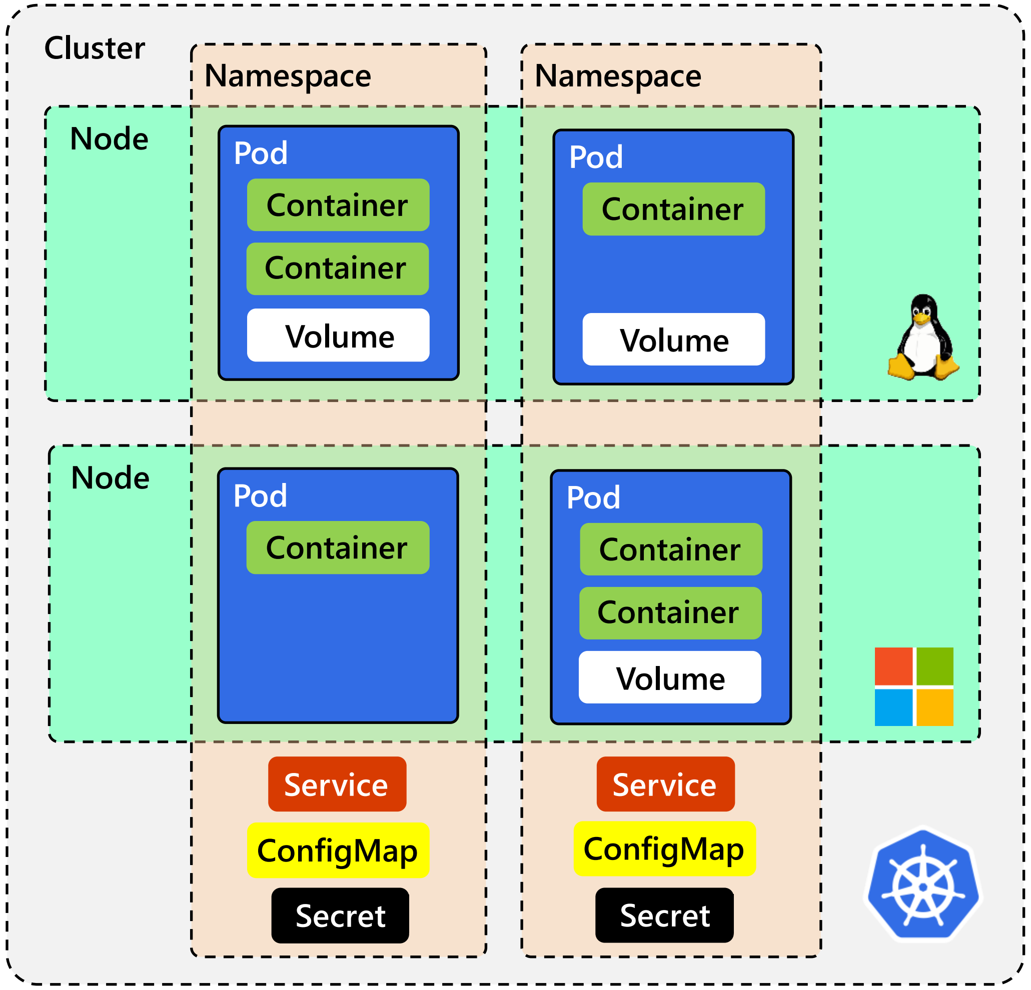
- **apiVersion** - Which API group and version of the API you're calling to create this object.
- **kind** - What kind (type) of object you want to create.
- **metadata** - Data that helps uniquely identify the object, including a name string, UID, and optional namespace.
- **spec** (most objects) – Specifies the desired state for the object.



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    tier: backend
    app: myapp
spec:
  containers:
    # Getting the latest nginx
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
          protocol: TCP
      env:
      - name: myvar
        value: abcd
      - name: myvar2
        value: abcd3
  nodeSelector:
    kubernetes.io/os: linux
```

Overview of Primary Kubernetes Resources

Let's look at how primary Kubernetes resources are related to each other.



- **Cluster** – A collection of machines networked together.
- **Node** – A single machine (physical or VM) in a cluster.
- **Pod** – A wrapper around one or more Containers. A Pod is scheduled on a Node.
- **Container** – A single containerized application instance, based on an image, managed by a Pod.
- **Volume** – A connection to external storage outside the Pod.
- **Service** – A network configuration directing traffic to a Pod.
- **ConfigMap/Secret** – An external configuration that can be referenced by multiple Pods.
- **Namespace** – A virtual partition to keep related resources together.

Optional - Kubernetes for Developers

View Kubernetes Internals for Developers Presentation

Kubectl (“kube cuttle” or “kube control”)

- **Kubectl** is the official command line utility used to interact with the Kubernetes cluster and its resources.
- Common commands include:
  - ***Get*** – Returns a list of objects. Can get all the details for a specific object in *yaml* or *json* format.
    - **Example**: Get a list of Nodes: `kubectl get nodes`
    - **Example**: Retrieve the YAML definition of a Pod from *etcd*: `kubectl get pod pod-123 -o yaml`
  - ***Describe*** – Displays a detailed description of an object and related information from other objects.
    - **Example**: Show detailed description of a service: `kubectl describe service/myservice`
  - ***Logs*** – Displays the *stdout* log from a container in a Pod:
    - **Example**: `kubectl logs pod-123 -c nginx`
  - ***Exec*** – Executes a command in container in a Pod:
    - **Example**: Get a list of Nodes: `kubectl exec -it pod-123 -c nginx -- bash`

Kubectl Apply and Create Commands

- **Kubectl** has two commands for creating objects in its database: ***Apply*** & ***Create***.
- ***Apply*** creates objects through a *declarative* syntax.
  - **Example**: Create or update any resource defined in a *yaml* manifest:

    ```
    kubectl apply –f mydep.yaml
    ```
  - You can edit the manifest file and repeat the ***apply*** command to update the object in the cluster.
- ***Create*** creates objects *imperatively*.
  - **Example**: Create a Deployment object imperatively using system defaults:

    ```
    kubectl create deployment mydep --image=nginx --replicas=2
    ```
  - Since there's no *yaml* file to update and reapply, you can update the object using the ***Edit*** command.

    ```
    kubectl edit deployment mydep
    ```
  - You can extract an existing object definition into a *yaml* file and update it later using the ***apply*** command:

    ```
    kubectl get deployment mydep -o yaml > mydeployment.yaml
    ```
  - You can also use ***create*** to generate a new *yaml* file for you, without creating the object in Kubernetes:

    ```
    kubectl create deployment mydep2 --image=nginx --dry-run -o yaml > mydep2.yaml
    ```

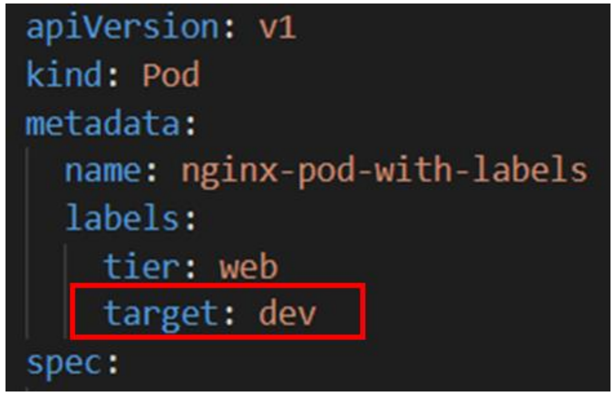
Kubectl Demo

Various Kubectl commands

Labels & Selectors

- **Labels**
  - Labels are declarative way to identify Kubernetes objects.
  - Simple key-value pair (not a list, no duplicates)
  - A Kubernetes object can have zero to many labels

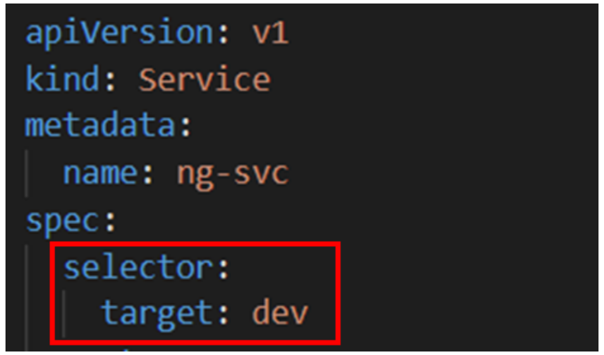
```
kubectl get pods --show-labels
```



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-with-labels
  labels:
    tier: web
    target: dev
spec:
```

- **Selectors**
  - Mechanism to filter for labels that match certain criteria or logic
  - Used by objects to find and associate to other objects
  - **Example:** Gets all Pods that have the ***target: dev*** label:

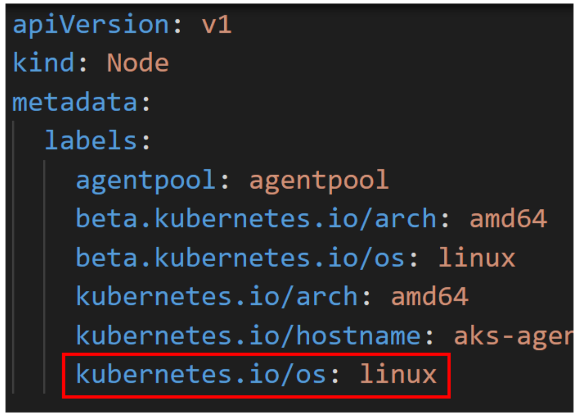
```
kubectl get pods --selector=target=dev
```



```
apiVersion: v1
kind: Service
metadata:
  name: ng-svc
spec:
  selector:
    target: dev
```

Using Well-known Labels and Node Selectors

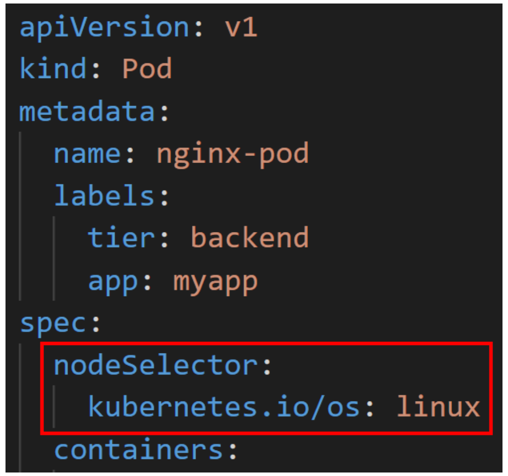
- When Worker Nodes are added to a cluster, they are expected to include certain well-known labels.
  - Kubernetes.io/arch
  - Kubernetes.io/os
  - etc.



```
apiVersion: v1
kind: Node
metadata:
  labels:
    agentpool: agentpool
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/os: linux
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: aks-ager
    kubernetes.io/os: linux
```

- You can use a ***Node Selector*** to tell the scheduler to put your workloads only on nodes that have a specific label.

- Labels and Selectors are used by many resources in Kubernetes to create associations between objects.



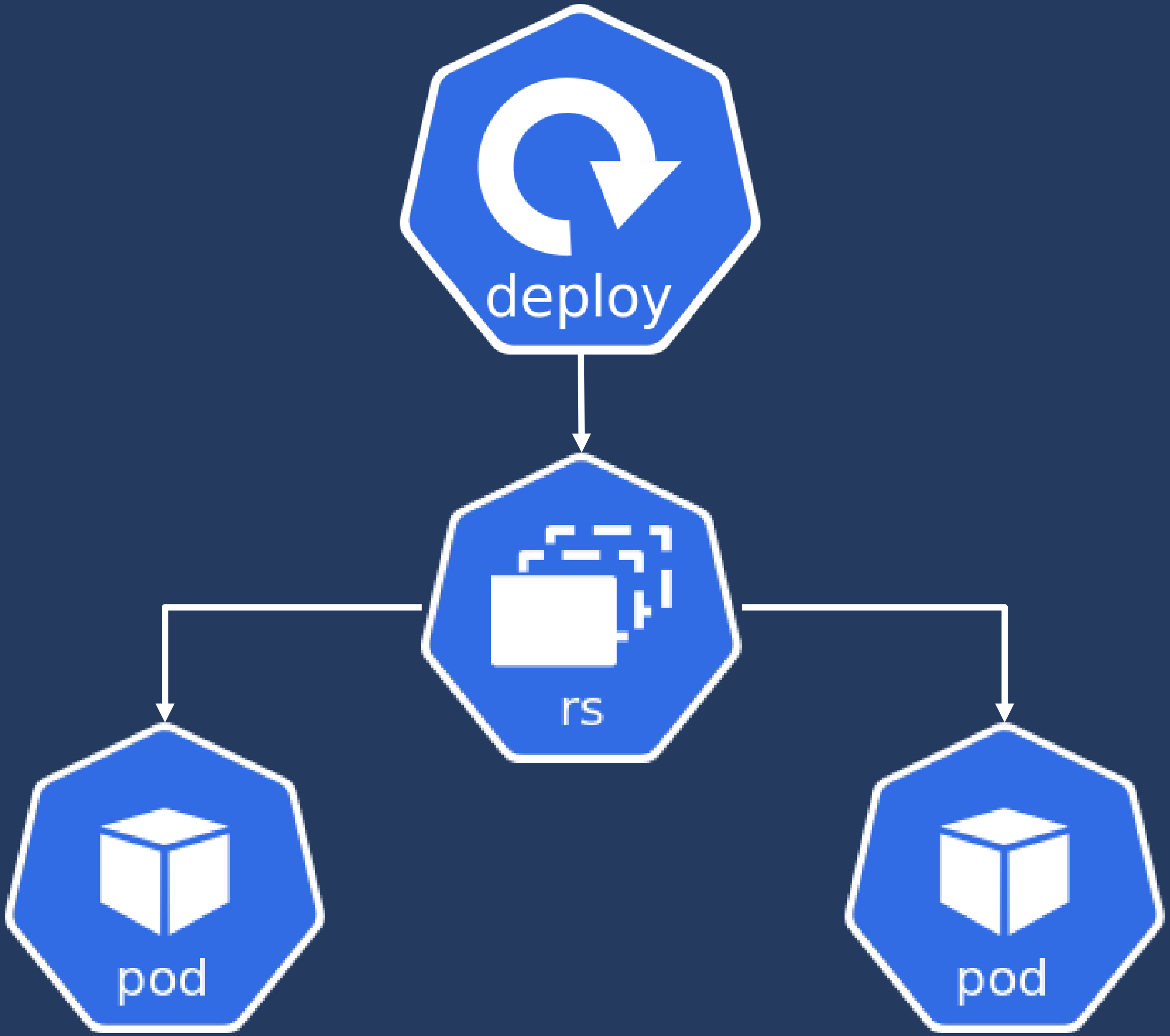
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    tier: backend
    app: myapp
spec:
  nodeSelector:
    kubernetes.io/os: linux
  containers:
```

Labels Demo

Working with Labels and Selectors

2

Kubernetes Core Workloads



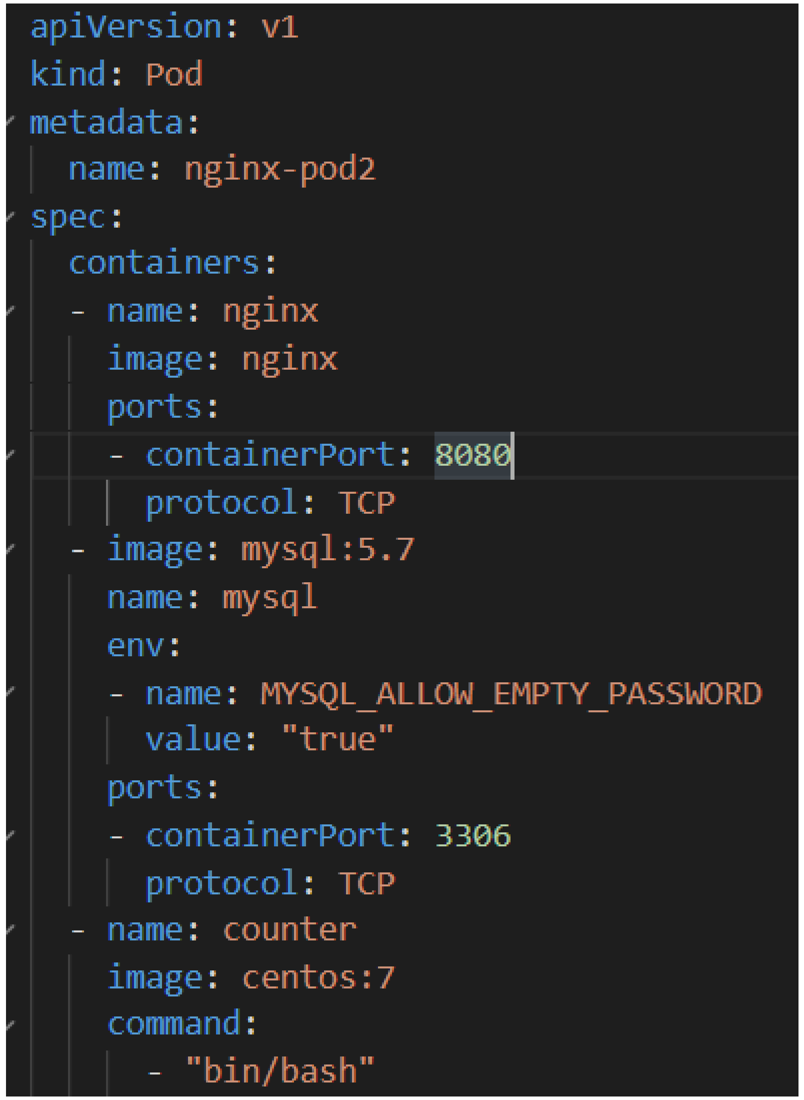
Pods

pod

- In Kubernetes, you never directly run a container, instead you run a **Pod**.
- A Pod is a "wrapper" around one or more containers.

Pods are NOT Containers

- Pods are the smallest deployable compute units you can create and manage in Kubernetes.
- A Pod can manage one or more containers, with shared storage (volumes) and network resources, and a specification for how to run the containers.
- Containers running in a Pod share the same IP and ports and communicate using native inter-process communication channels or *localhost*.
- Pods are immutable - if any change is made to the Pod specification (***spec***), a new Pod is created and then the old Pod is deleted.

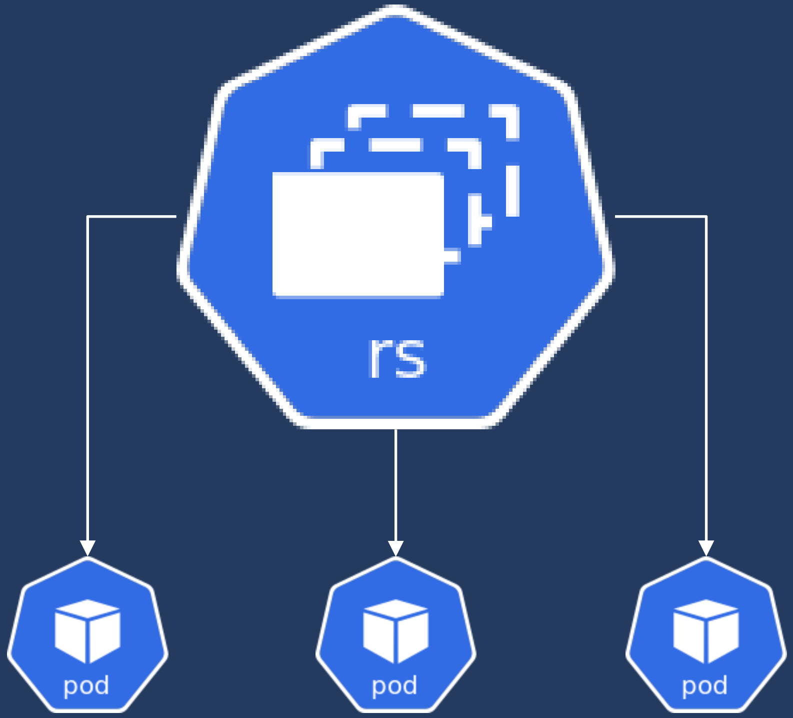


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod2
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 8080
      protocol: TCP
  - image: mysql:5.7
    name: mysql
    env:
    - name: MYSQL_ALLOW_EMPTY_PASSWORD
      value: "true"
    ports:
    - containerPort: 3306
      protocol: TCP
  - name: counter
    image: centos:7
    command:
      - "bin/bash"
```

Pod Demo

Create and manage a Pod

Replica Sets

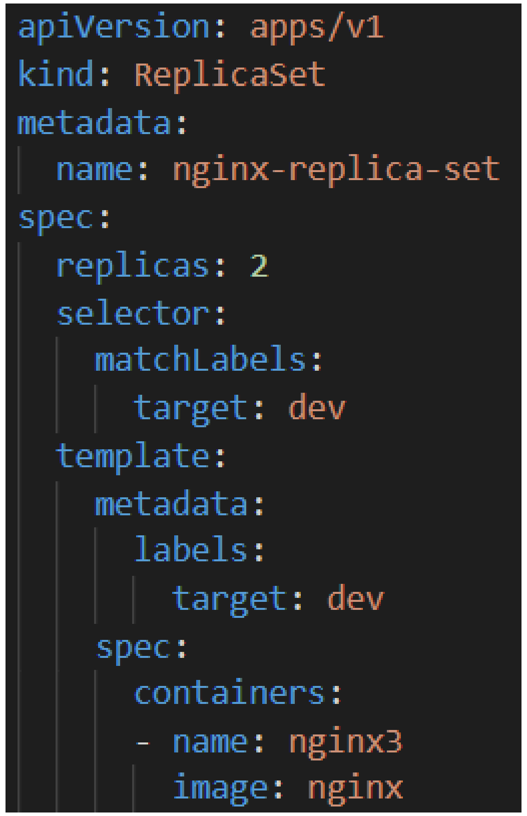


- Containers are created by ***Controllers*** through Pods.
- A **ReplicaSet** is a controller whose purpose is to guarantee the availability of a specified number of identical Pods.

replicas set the number of Pods to create

selector finds the template

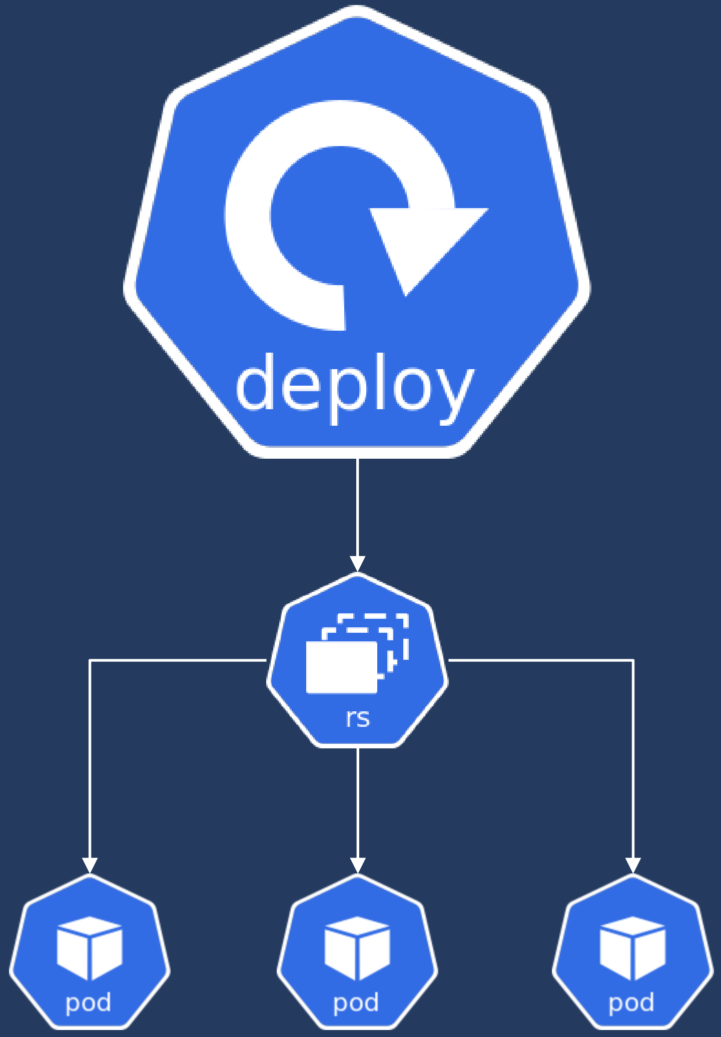
template defines the PodSpec:



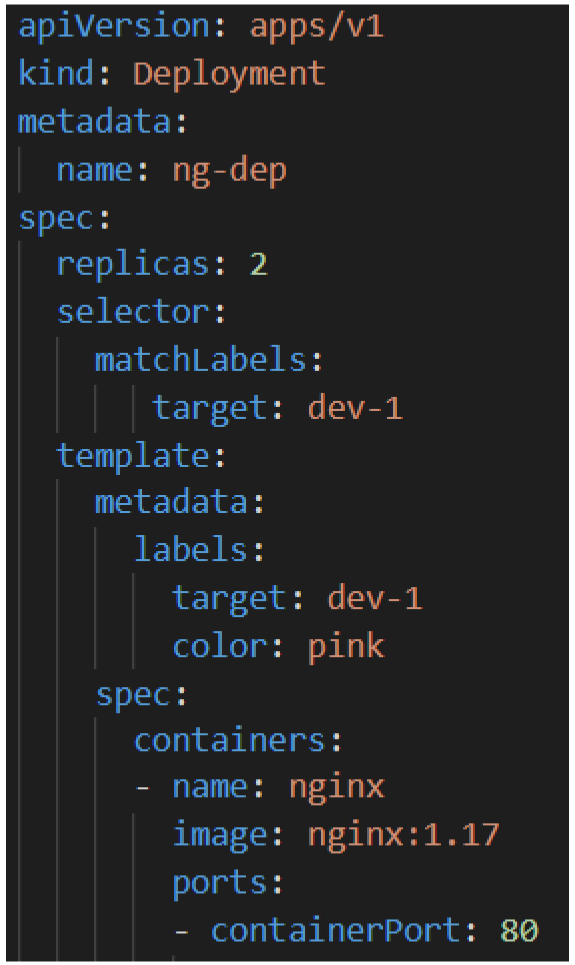
```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replica-set
spec:
  replicas: 2
  selector:
    matchLabels:
      target: dev
  template:
    metadata:
      labels:
        target: dev
    spec:
      containers:
      - name: nginx3
        image: nginx
```

- However, a **ReplicaSet** is too low-level to work with directly.

Deployments



- A **Deployment** is a higher-level controller that manages **ReplicaSets** and provides declarative updates to Pods along with a lot of other useful features.
- You describe a *desired state* in a **Deployment**, and the Deployment Controller changes the actual state to the desired state at a controlled rate.
- Deployments provide fine-grained control over how and when a new pod version is rolled out as well as rolled back to a previous state



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ng-dep
spec:
  replicas: 2
  selector:
    matchLabels:
      target: dev-1
  template:
    metadata:
      labels:
        target: dev-1
        color: pink
    spec:
      containers:
      - name: nginx
        image: nginx:1.17
        ports:
        - containerPort: 80
```

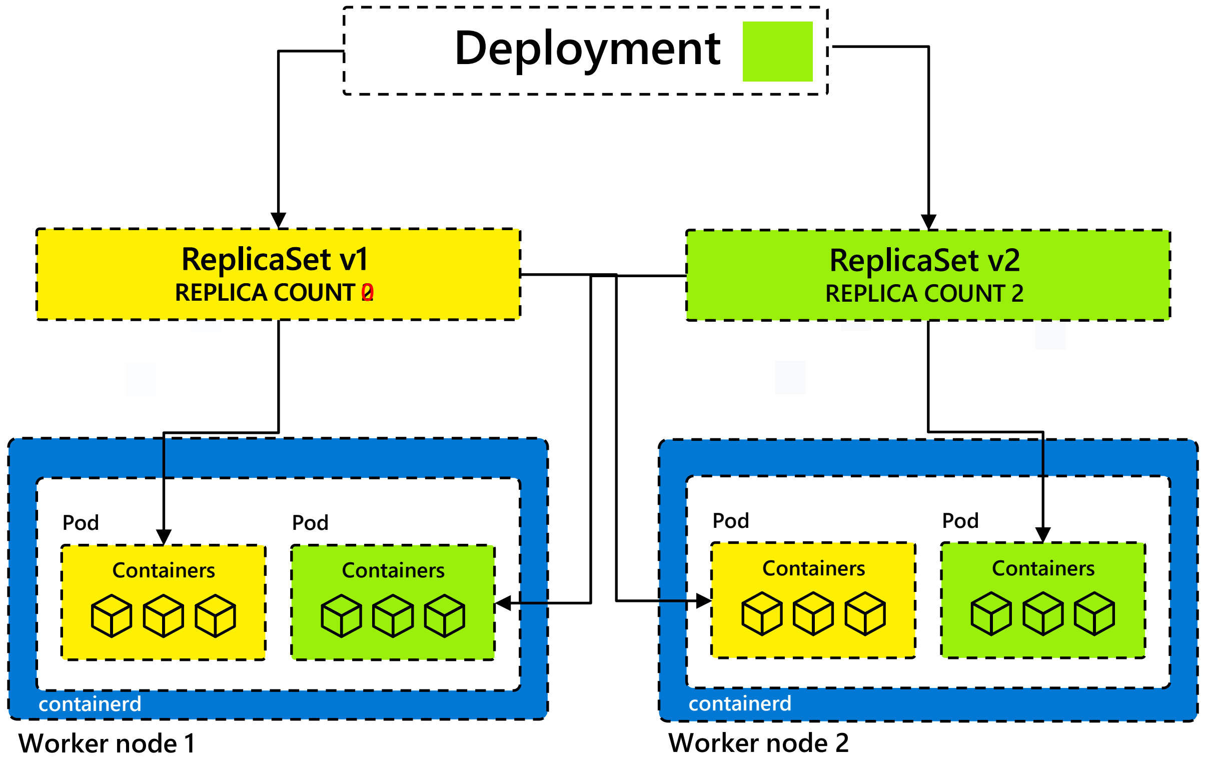
Deployment at Work

1. Let's say we are running a deployment with 2 replicas of a pod. For illustration, our deployment is designated with a yellow background.

2. Developer makes the change to the deployment image tag and to sets the background to green

3. The Deployment creates a ReplicaSet v2, which will create 2 new Pods with the updated image.

4. Once the new Pods are up and running, the Pods attached to ReplicaSet v1 are deleted.

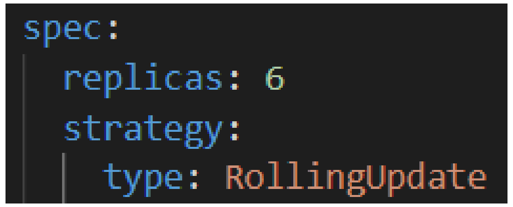


NOTE: Notice that ReplicaSet v1 remains.

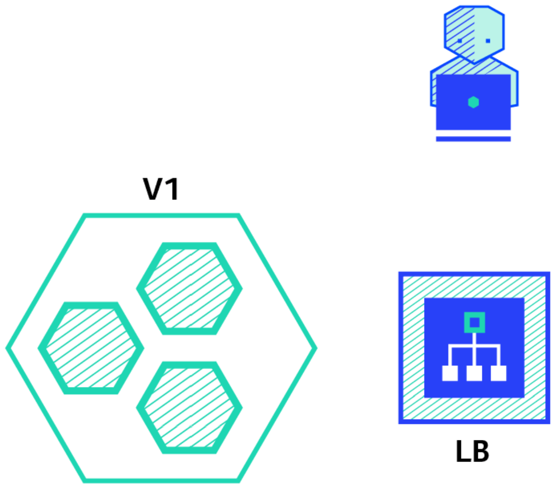
Natively Supported Deployment Strategies

A deployment strategy is a way to change or upgrade an application. The two directly supported Kubernetes deployment strategies are:

Rolling Update - Consists of *slowly* rolling out a new version of an application by replacing instances one after the old version until all the instances are rolled out. This is the *default* strategy for Kubernetes Deployments when none is specified.

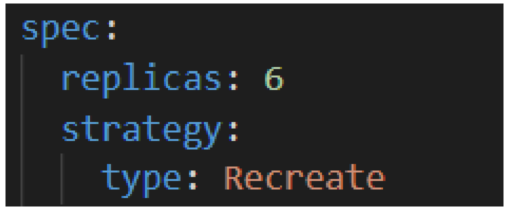


```
spec:
  replicas: 6
  strategy:
    type: RollingUpdate
```

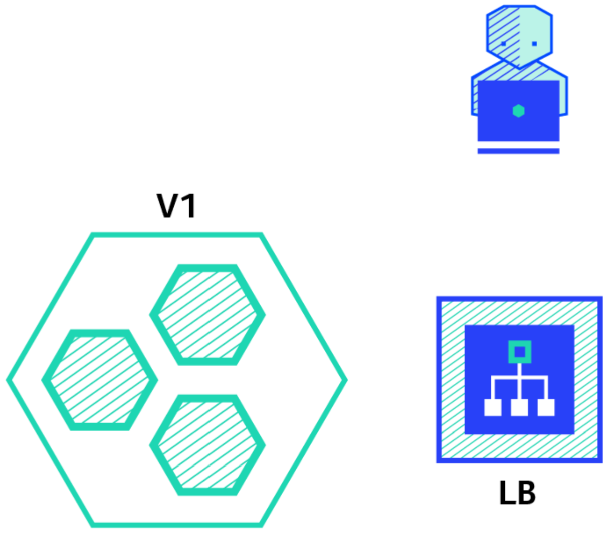


Natively Supported Deployment Strategies

Recreate - Consists of *shutting down* all instances of the current version, then deploying the new version. This technique implies downtime of the service that depends on both shutdown and boot duration of the application.



```
spec:
  replicas: 6
  strategy:
    type: Recreate
```



Deployments Demo

Working with Deployments

Kubectl Imperative Commands

In addition to *Create*, **Kubectl** supports other imperative commands to interact with deployments (and other objects) already in the database.

- **Example**: Change the image in a deployment without editing and reapplying the *yaml*:
  ```
  kubectl set image deployment/mydep nginx=nginx:1.18
  ```
- **Example:** View the deployment rollout history of a deployment:
  ```
  kubectl rollout history deployment/mydep
  ```
- **Example:** Change the number of replicas of a deployment without editing the *yaml*:
  ```
  kubectl scale --replicas=3 deployment/mydep
  ```
- **Example:** Add a label to a deployment without editing the *yaml*:
  ```
  kubectl label deployment/mydep newlabel=myvalue
  ```
- **Example:** Listen on port 5000 on the local machine and forward to port 6000 on the Pod:
  ```
  kubectl port-forward mypod 5000:6000
  ```

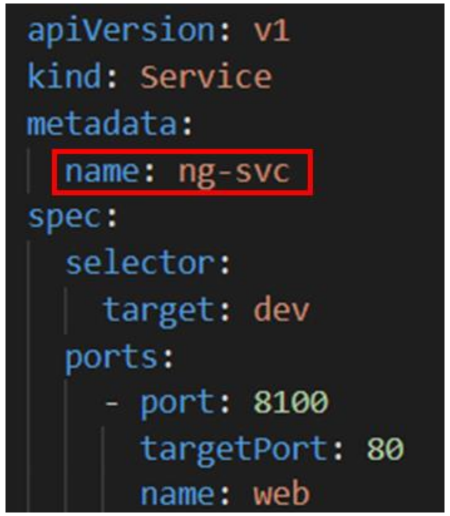
Additional examples for using the Kubectl command can be found on the Kubectl Cheat Sheet.

Services

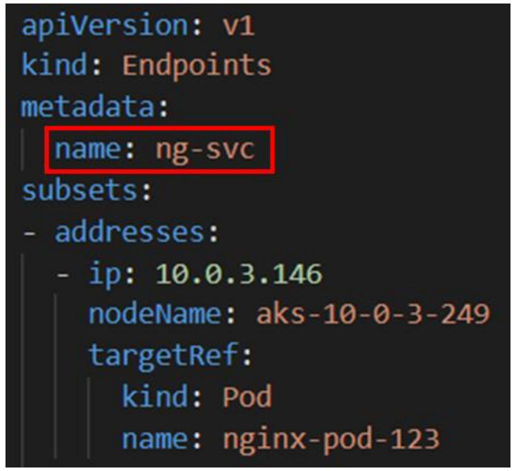
3

What is a Service?

- A ***Service*** is an abstraction that defines a logical set of loosely-coupled PODs and a policy by which to access them as a network service.
  - Use *selectors* to define which pods to include.
  - Services load balance traffic to Pod replicas (Layer 4)
  - Maps external ports to target (container) ports
  - Exposes Pods to traffic outside the cluster.
- Kubernetes creates an ***Endpoints*** object with the same name as the ***Service***.
- As Pods are deleted and replaced, the ***Endpoints*** object is automatically updated with the new Pods' IP addresses, as soon as those Pods go into a ***Ready (Running)*** state.
- This allows the Service's load balancer to immediately know which Pods it has to choose from when routing incoming traffic.



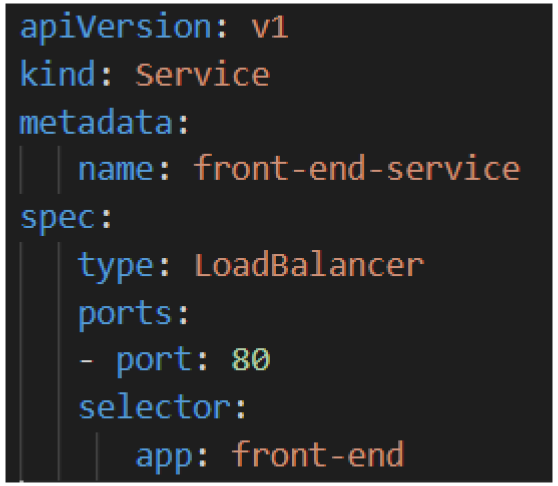
```yaml
apiVersion: v1
kind: Service
metadata:
  name: ng-svc
spec:
  selector:
    target: dev
  ports:
    - port: 8100
      targetPort: 80
      name: web
```



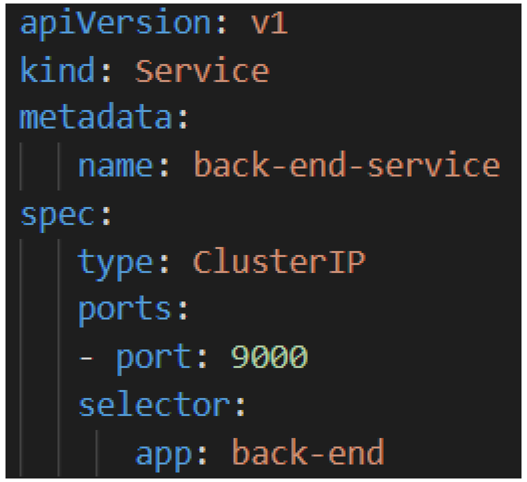
```yaml
apiVersion: v1
kind: Endpoints
metadata:
  name: ng-svc
subsets:
- addresses:
  - ip: 10.0.3.146
    nodeName: aks-10-0-3-249
    targetRef:
      kind: Pod
      name: nginx-pod-123
```

Services at Work

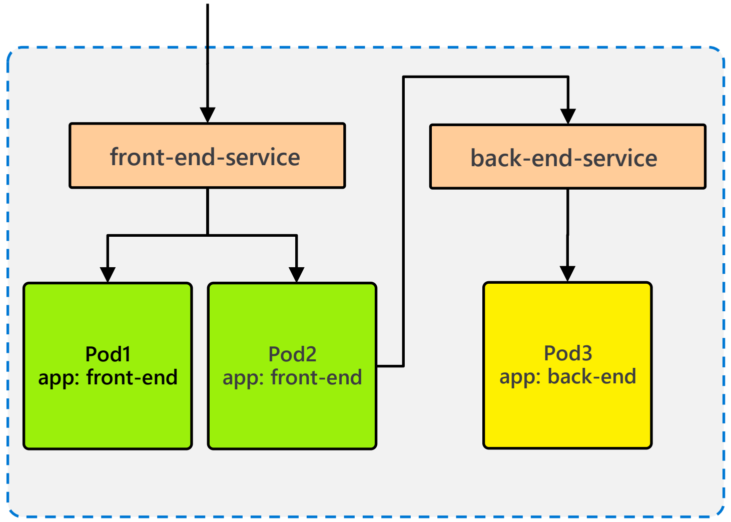
- You don't need to modify your application to use an unfamiliar service discovery mechanism.
- Unlike Pods, Service names/IPs don’t change, so they can be used for reliable routing configurations.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: front-end-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: front-end
```

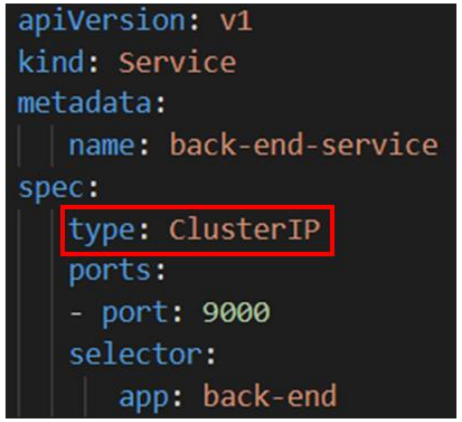


```yaml
apiVersion: v1
kind: Service
metadata:
  name: back-end-service
spec:
  type: ClusterIP
  ports:
  - port: 9000
  selector:
    app: back-end
```

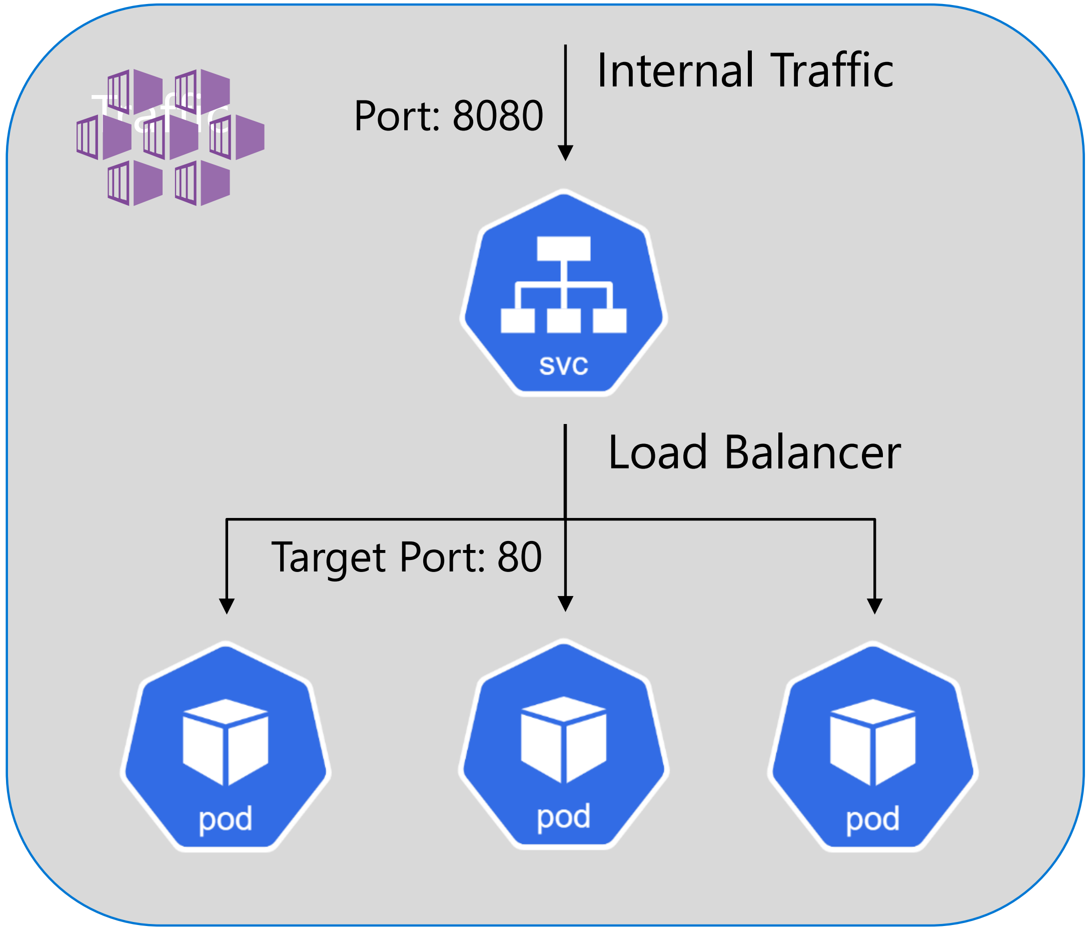


Service Types – ClusterIP

- **ClusterIP** – An internal IP address for use within the Kubernetes cluster.
- Used for internal communications between workloads within the cluster.
- Default service type when one is not specified.



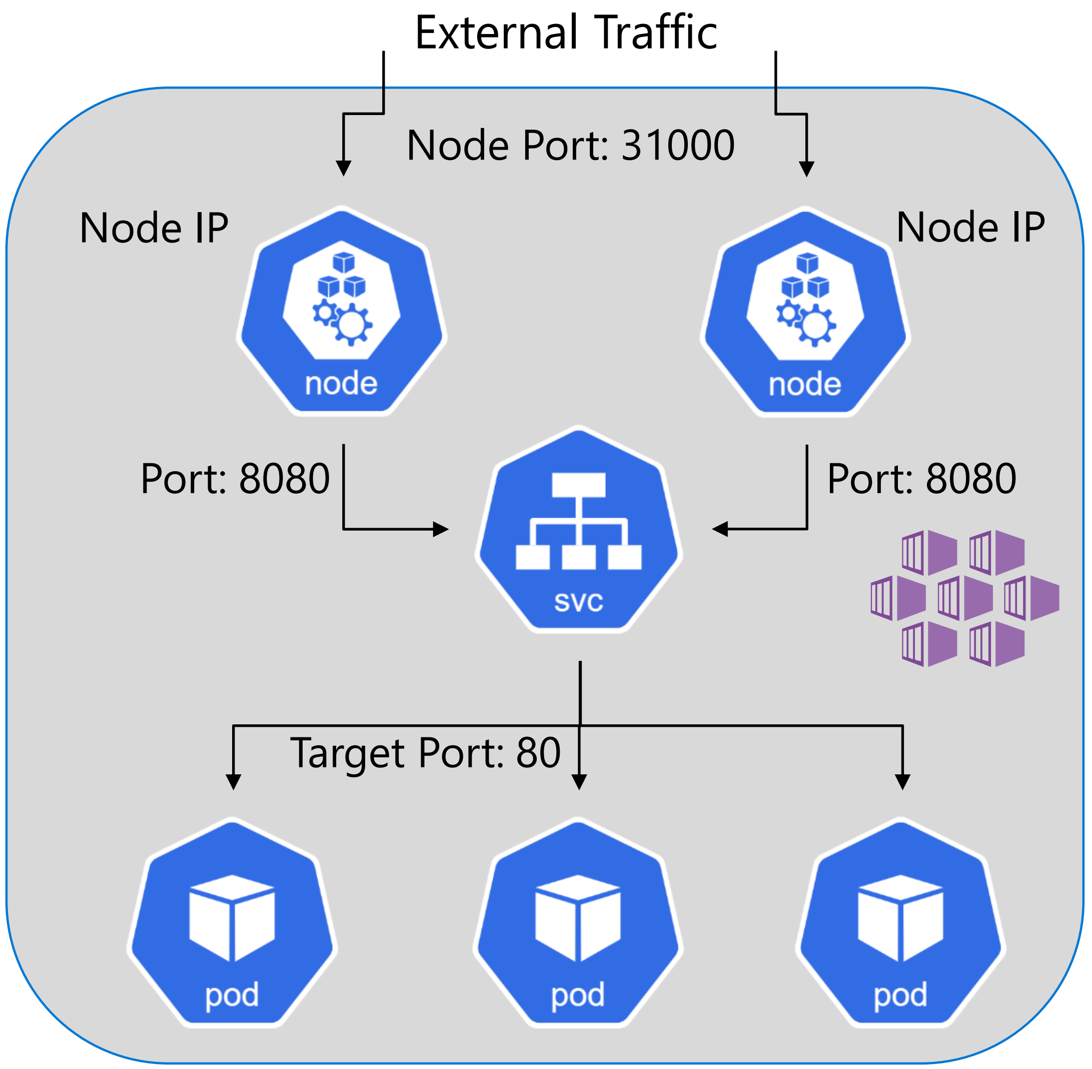
```yaml
apiVersion: v1
kind: Service
metadata:
  name: back-end-service
spec:
  type: ClusterIP
  ports:
  - port: 9000
  selector:
    app: back-end
```



Service Types – NodePort

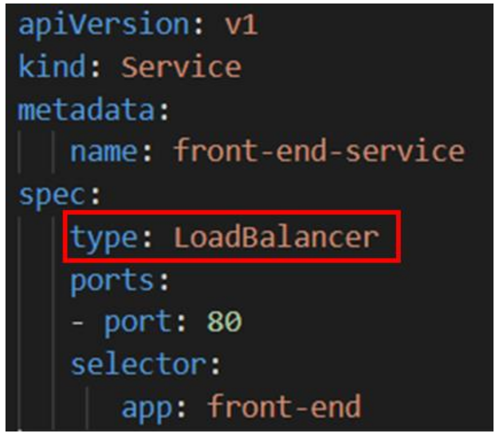
- **NodePort** - Creates a port mapping to the underlying Node, which allows external access directly to Pods by using the Node IP address (any Node's IP) and the Node Port.
- Used in on-premise Kubernetes clusters to expose services for external access.
- Only practical in AKS clusters when the Nodes' IP addresses are externally accessible, such as for Game Servers. Can be configured during cluster creation using this parameter:

```
--enable-node-public-ip
```

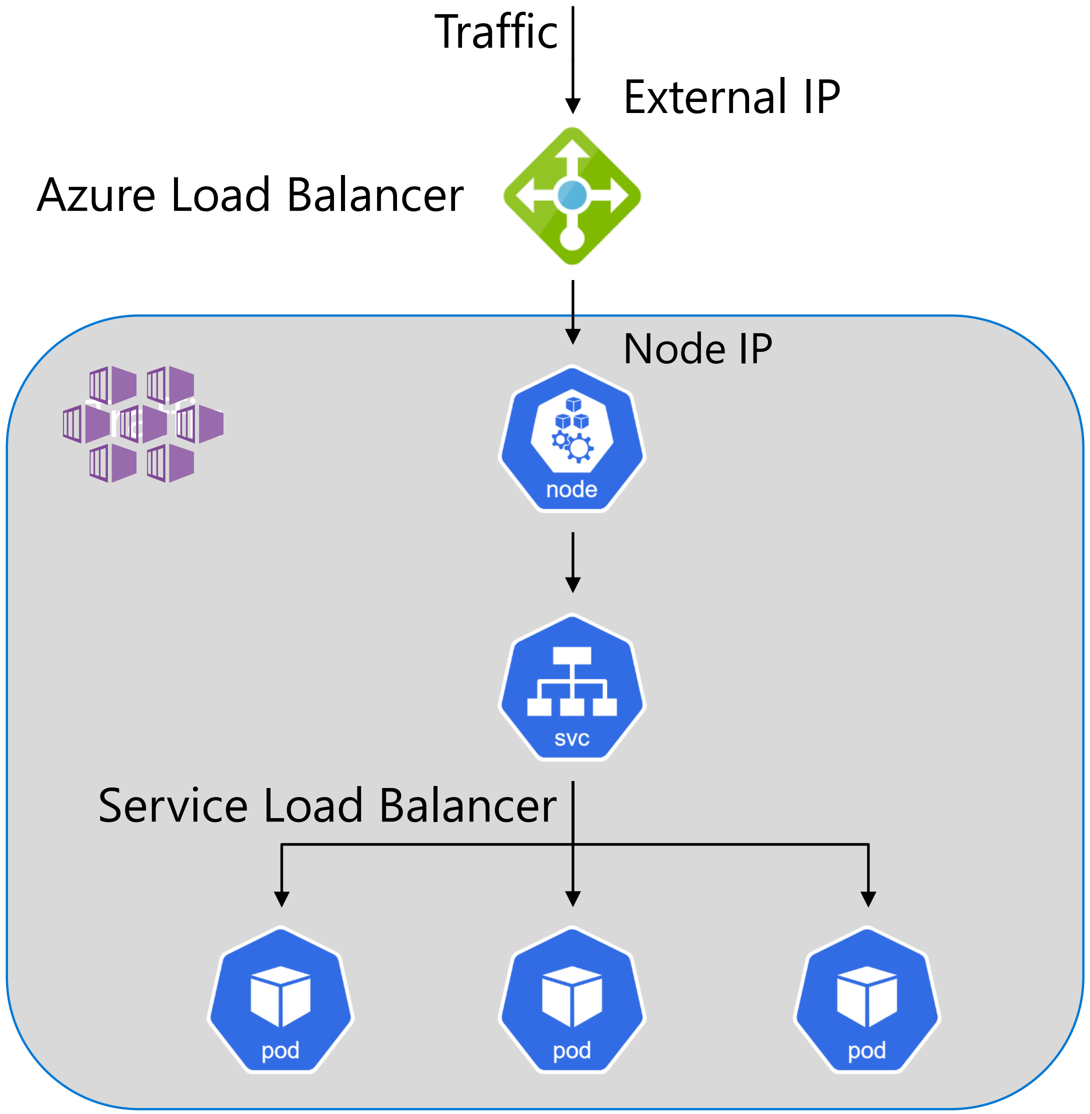


Service Types – LoadBalancer

- **LoadBalancer** – Requests an external IP address (public by default) from the Azure Load Balancer.
- Azure creates an IP Address resource and adds it to the load balancer backend pool.
- Allow external traffic to reach Pods in the cluster.

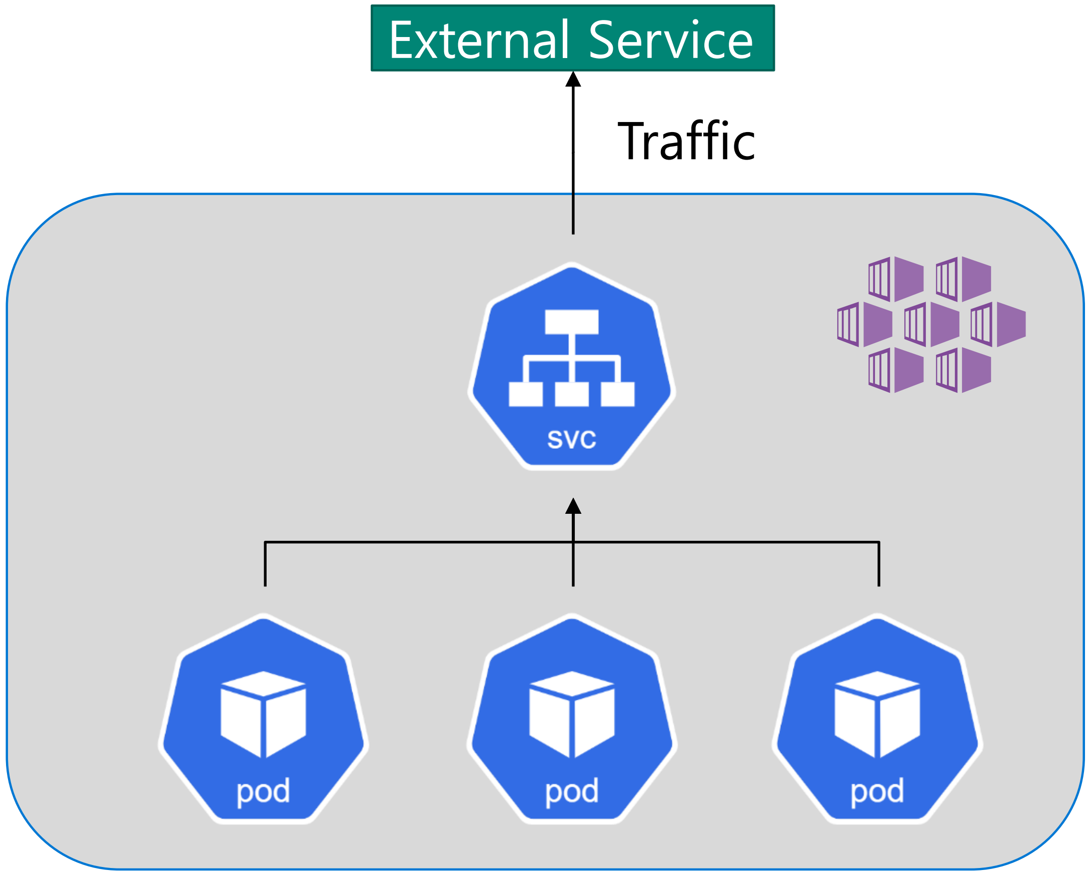


```
apiVersion: v1
kind: Service
metadata:
  name: front-end-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: front-end
```



Service Types – ExternalName

- **ExternalName** – A special case of Service that does not have selectors, because it's used for external access.
- An **ExternalName** maps a Service to a DNS name, for Pods to have consistent routes to external resources.



Services Demo

Working with Services

4

Config Maps and Secrets

cm

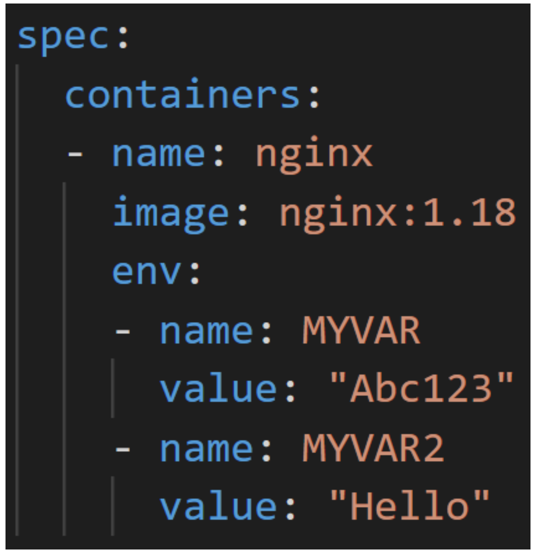
secret

Pod Environment Variables

- When working with containers in Docker, you can define environment variables when you create your containers, either individually or by specifying a file:

```
docker run -e MYVAR1 --env MYVAR2=foo --env-file ./env.list ubuntu
```

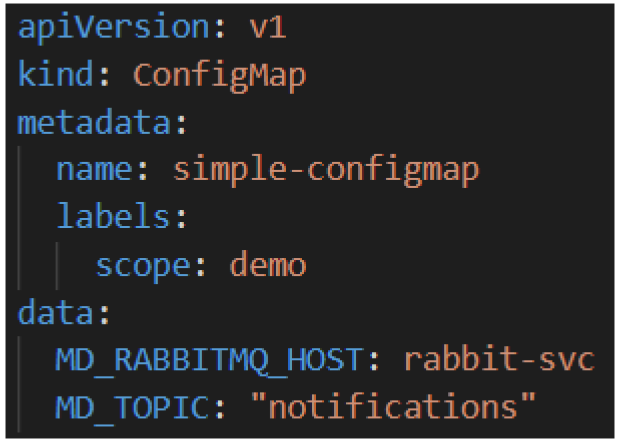
- When creating Pods, you can specify environment variables as a list of *name/value* pairs in the container's specification.
- If you want to use an entire file full of environment variables, you can create a ***ConfigMap*** and reference all or part of it in the container's specification.



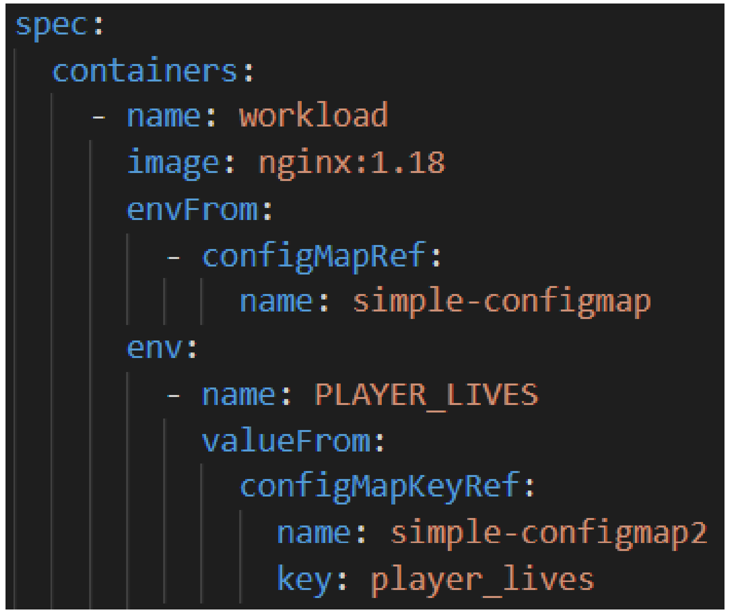
```yaml
spec:
  containers:
  - name: nginx
    image: nginx:1.18
    env:
    - name: MYVAR
      value: "Abc123"
    - name: MYVAR2
      value: "Hello"
```

Config Maps

- A ***ConfigMap*** is used to store non-confidential data as key-value pairs.
- They allows you to decouple environment-specific configuration from container instances.
- Containers can consume ***ConfigMaps*** as environment variables.
- Containers can use an entire ***ConfigMap*** or only selected values.
- ***ConfigMaps*** can be referenced by multiple Containers in different Pods.



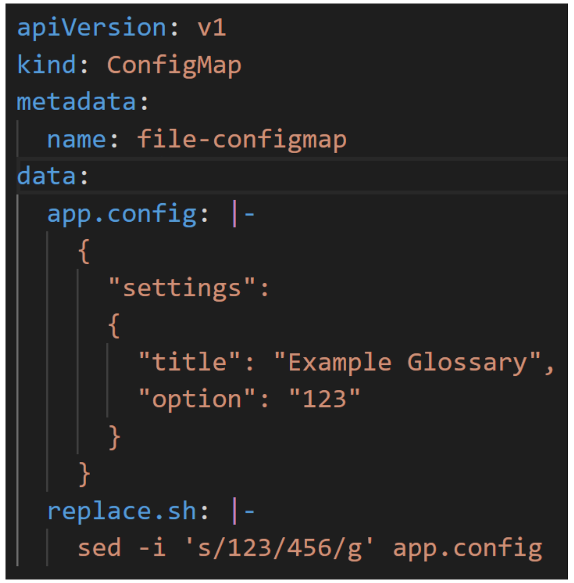
```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: simple-configmap
  labels:
    scope: demo
data:
  MD_RABBITMQ_HOST: rabbit-svc
  MD_TOPIC: "notifications"
```



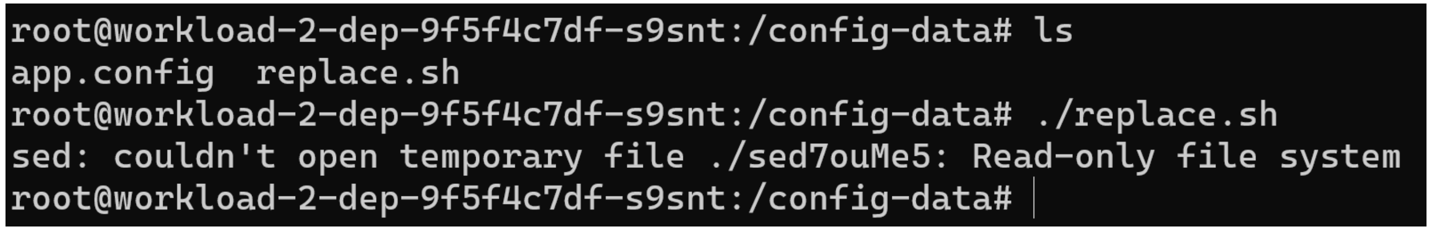
```yaml
spec:
  containers:
    - name: workload
      image: nginx:1.18
      envFrom:
        - configMapRef:
            name: simple-configmap
      env:
        - name: PLAYER_LIVES
          valueFrom:
            configMapKeyRef:
              name: simple-configmap2
              key: player_lives
```

ConfigMaps Mounted as Volumes

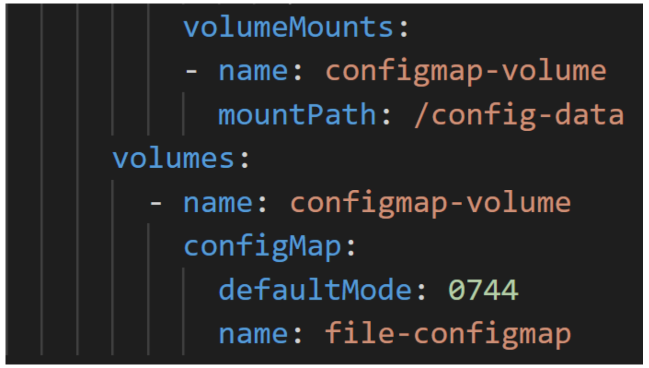
- ***ConfigMaps*** can also contain entire file definitions.
- These ***ConfigMaps*** can be attached as Volumes in a Pods and mounted as *read-only* folders in containers.
- Each entry in a ***ConfigMaps*** is mounted as a separate file.
- If scripts are defined in ***ConfigMaps***, they can be automatically mounted with executable permissions, using the **defaultMode** setting.



```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: file-configmap
data:
  app.config: |-
    {
      "settings":
      {
        "title": "Example Glossary",
        "option": "123"
      }
    }
  replace.sh: |-
    sed -i 's/123/456/g' app.config
```



```
root@workload-2-dep-9f5f4c7df-s9snt:/config-data# ls
app.config  replace.sh
root@workload-2-dep-9f5f4c7df-s9snt:/config-data# ./replace.sh
sed: couldn't open temporary file ./sed7ouMe5: Read-only file system
root@workload-2-dep-9f5f4c7df-s9snt:/config-data#
```



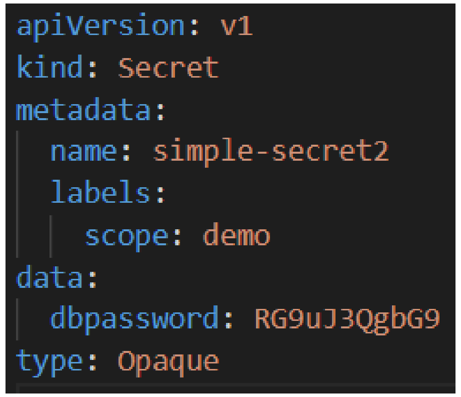
```yaml
    volumeMounts:
    - name: configmap-volume
      mountPath: /config-data
volumes:
  - name: configmap-volume
    configMap:
      defaultMode: 0744
      name: file-configmap
```

ConfigMaps Demo

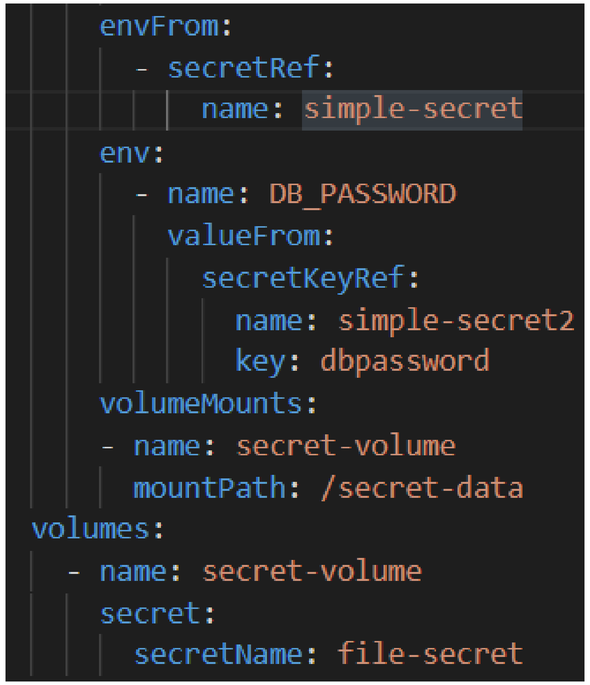
Working with ConfigMaps

Secrets

- ***Secrets*** are the same as ***ConfigMaps***, except the data/files they hold is *encoded*.
- They can be used to store and manage sensitive information, such as passwords, OAuth tokens, and *ssh* keys.
- Storing confidential information in a Secret is safer and more flexible than setting it verbatim in a Pod definition or in a container image.
- Other Kubernetes resources, like ***Ingresses*** and ***Persistent Volumes***, also use ***Secrets*** as part of their configuration.



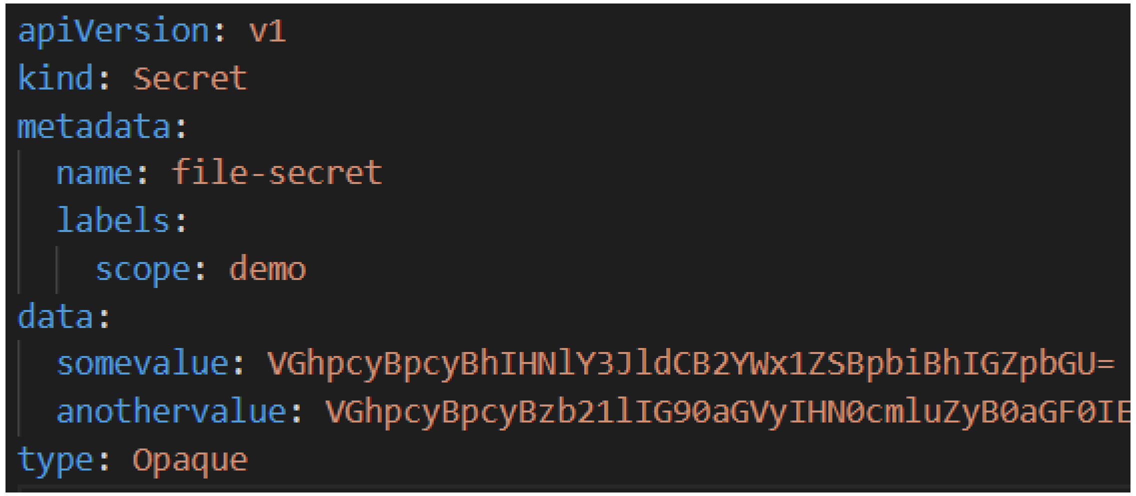
```yaml
apiVersion: v1
kind: Secret
metadata:
  name: simple-secret2
  labels:
    scope: demo
data:
  dbpassword: RG9uJ3QgbG9
type: Opaque
```



```yaml
    envFrom:
      - secretRef:
          name: simple-secret
    env:
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: simple-secret2
            key: dbpassword
    volumeMounts:
    - name: secret-volume
      mountPath: /secret-data
volumes:
  - name: secret-volume
    secret:
      secretName: file-secret
```

Secrets are Encoded, NOT Encrypted

- Kubernetes ***Secrets*** are, by default, stored as *<u>unencrypted base64-encoded strings</u>*.
- ***Secrets*** can be viewed as plain text by anyone with API access or anymore who can execute a *printenv* command in a container.



```yaml
apiVersion: v1
kind: Secret
metadata:
  name: file-secret
  labels:
    scope: demo
data:
  somevalue: VGhpcyBpcyBhIHNlY3JldCB2YWx1ZSBpbiBhIGZpbGU=
  anothervalue: VGhpcyBpcyBzb21lIG90aGVyIHN0cmluZyB0aGF0IE
type: Opaque
```

- It's recommended that you Enable or configure RBAC rules that restrict reading and writing of Secrets. Or inject, as needed, from Azure Key Vault.
- You can generate a ***Secret*** object definition from literal values and output it directly to a *yaml* file, without creating the object in the Kubernetes database.

```
kubectl create secret generic test-secret --from-file=tls=/path/tls.key \
     --from-literal=username=testuser -o yaml --dry-run > mysecret.yaml
```

Secrets Demo

Working with Secrets

5

Namespaces

Kubernetes Namespaces

- Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.
- Namespaces are intended for use in environments with many users spread across multiple teams, or projects.
- Namespaces provide a scope for resources. Resource names can be reused across namespaces.
- Get a list of Namespaces:

```
kubectl get namespaces
```

- Get a list of Pods in a namespace:

```
kubectl get pods –n mynamespace
```

Use Namespaces to Separate Resources

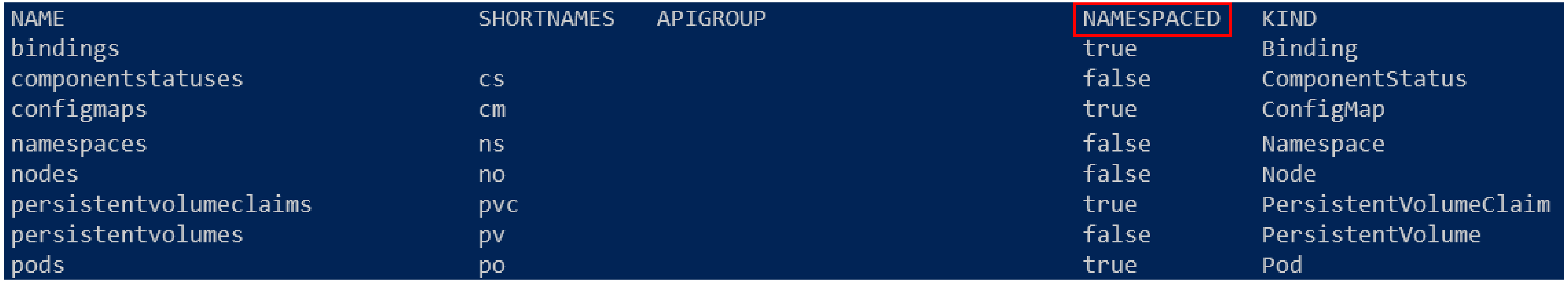
- Namespaces be used to segment resources by:
  - **Environments** – Ex: *Development* resources can be in one namespace, *Staging* in another
  - **Applications** – Ex: *Shopping* cart microservices in one namespace, *Marketing* in another
  - **Developers** – Ex: Each developer can have their own, isolated workspace.
    - User permissions/resources can be limited by namespace.

- To access resources across namespaces, use their FDQN:
  - Mysql.connect(“db-service.**othernamespace**.svc.cluster.local”)

Not all Resources use Namespaces

- Most resources are scoped to namespaces, but some are available across the entire cluster.
- Get a list of available namespace and cluster-scoped resources:

```
kubectl api-resources
```



```
NAME                    SHORTNAMES   APIGROUP              NAMESPACED   KIND
bindings                                                   true         Binding
componentstatuses       cs                                 false        ComponentStatus
configmaps              cm                                 true         ConfigMap
namespaces              ns                                 false        Namespace
nodes                   no                                 false        Node
persistentvolumeclaims  pvc                                true         PersistentVolumeClaim
persistentvolumes       pv                                 false        PersistentVolume
pods                    po                                 true         Pod
```

- Filter list of resources by namespaced scope:

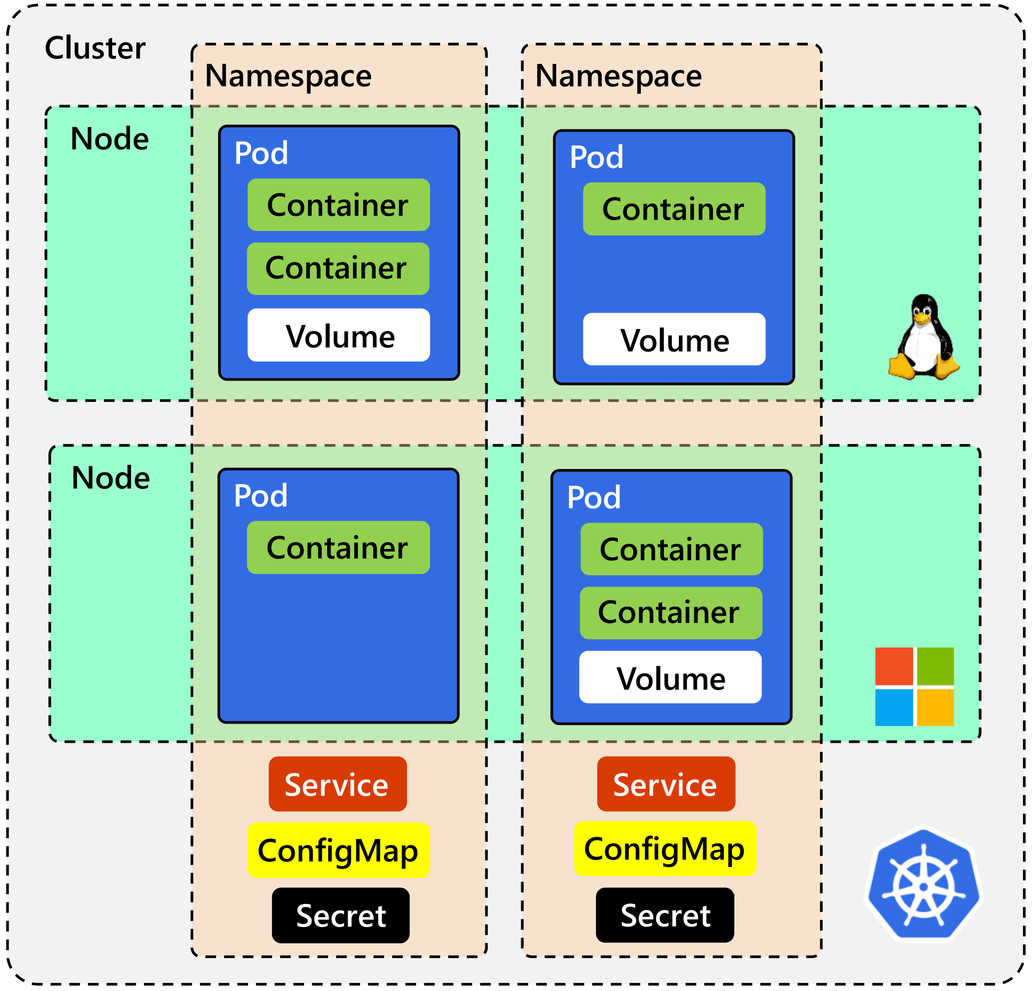
```
kubectl api-resources --namespaced=true
```

Namespaces Demo

Working with Namespaces

Review of Primary Kubernetes Resources

To review the relationships between primary Kubernetes resources:



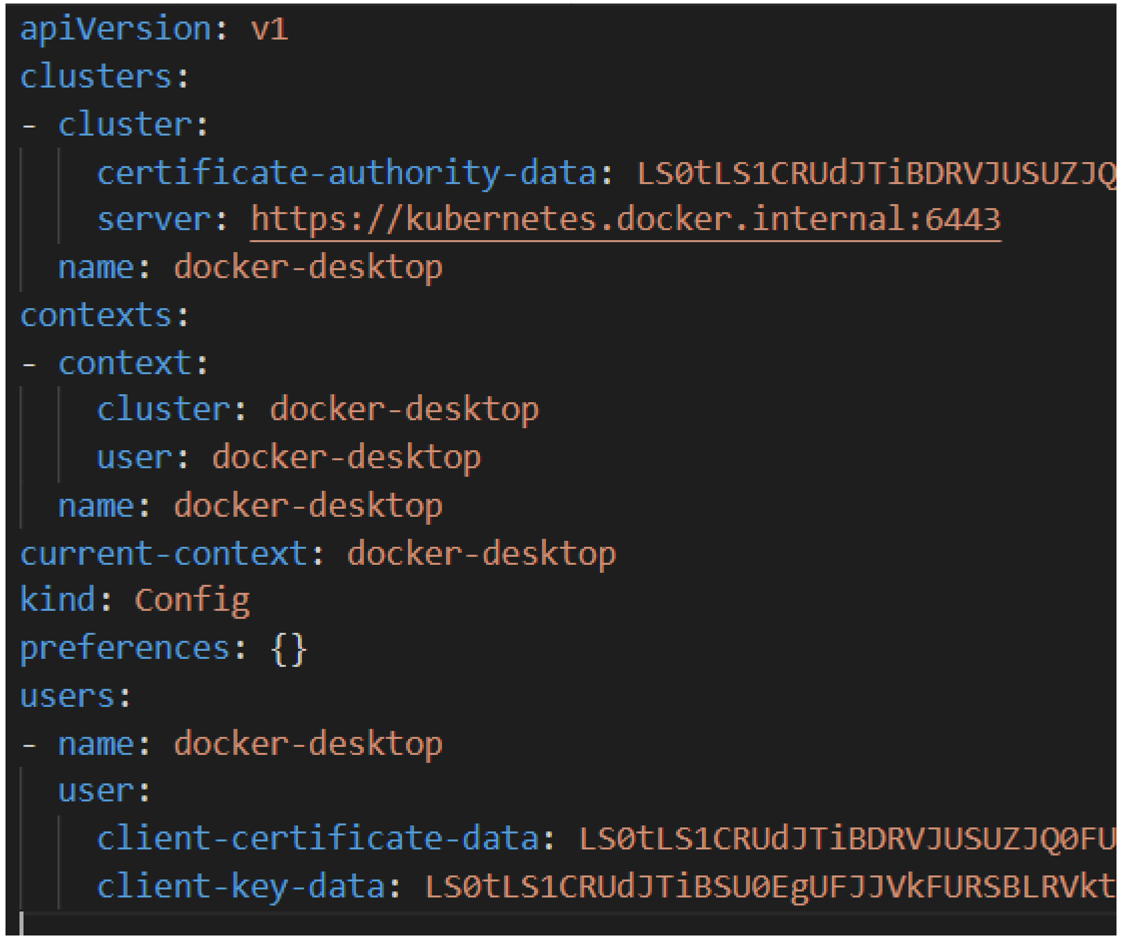
- **Cluster** – A collection of machines networked together.
- **Node** – A single machine (physical or VM) in a cluster.
- **Pod** – A wrapper around one or more Containers. A Pod is scheduled on a Node.
- **Container** – A single containerized application instance based on an image, managed by a Pod.
- **Volume** – A connection to external storage outside the Pod.
- **Service** – A network configuration directing traffic to a Pod.
- **ConfigMap/Secret** – An external configuration that can be referenced by multiple Pods.
- **Namespace** – A virtual partition to keep related resources together.

KubeConfig

6

KubeConfig

- The **KubeConfig** file organizes information about clusters, users, namespaces, and authentication mechanisms.
- Kubectl uses KubeConfig files to find the information it needs to choose a cluster and communicate with the API server of a cluster.
- By default, the KubeConfig is located in */usr/.kube* folder and is called *config*.
- A *context* element in a KubeConfig file is used to group three parameters together under a convenient name: cluster, namespace (default if not present) and user.
- Kubectl uses the *current-context* to determine which cluster to connect to.



```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ
    server: https://kubernetes.docker.internal:6443
  name: docker-desktop
contexts:
- context:
    cluster: docker-desktop
    user: docker-desktop
  name: docker-desktop
current-context: docker-desktop
kind: Config
preferences: {}
users:
- name: docker-desktop
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FU
    client-key-data: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVkt
```

KubeConfig Demo

Reviewing the Contents of KubeConfig

Lab – Module 1

Create Kubernetes Objects

Microsoft

Thank you