
Intro to Kubernetes

Deployments

What is a Deployment?

Kubernetes controller optimal for stateless applications

- Deployments allow you to declaratively manage pods, including replication
- Deployments support
  - Creating, rolling out, and rolling back changes to homogeneous set of pods
  - Scaling set of pods out and back declaratively
- Deployments include
  - Implicit Replica Set controller to handle pod replicas
  - Template spec of pods to be created and managed – no need to separately create pods
- Deployments used for web applications, mobile back-ends, API's

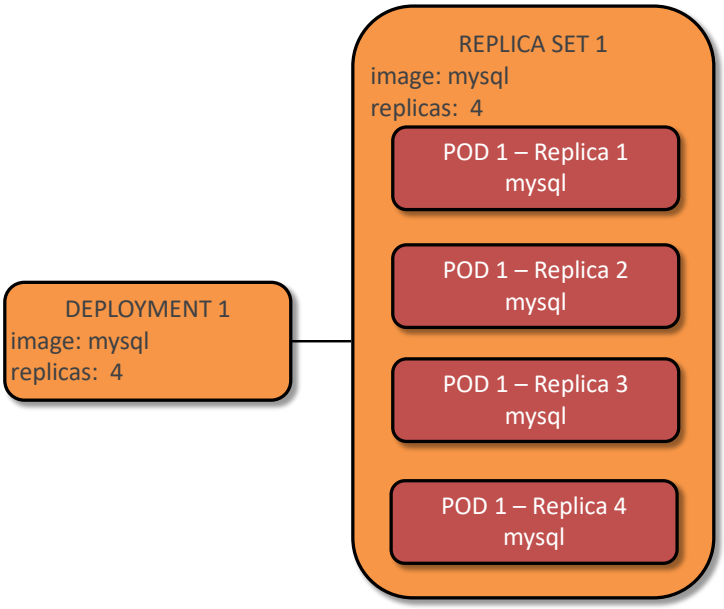
What if my Application isn't Stateless?

- Kubernetes provides other controller objects for applications that need different deployment schemes
- **StatefulSets** (previously PetSets) control deployment of pods for applications that need more stable deployment contexts
  - Pods in StatefulSets have unique ordinal, stable network identity and stable storage using persistent volumes
  - When pods are deployed, they are created in sequence of ordinals 1..N
  - Pod N must be running and ready before Pod N+1 is deployed
  - When pods are destroyed, they are terminated in reverse sequence N..1
- **DaemonSets** ensure that a replica of a specified pod is running on every node (or every selected node) in the cluster
- **Jobs** manage sets of pods where N must run to successful completion

Deployments Control ReplicaSet Controllers

Definition of how many replicated Pods should exist

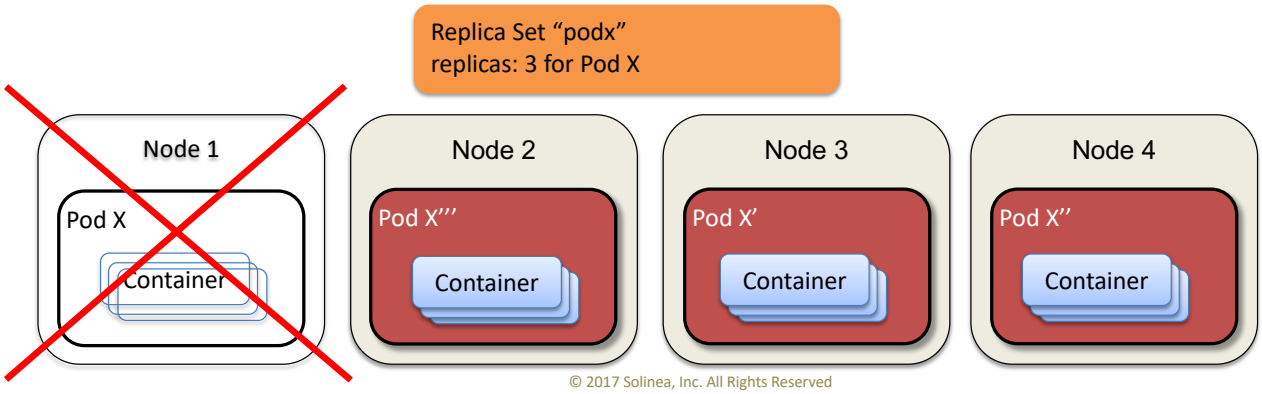
- Deployment creates and manages a Replica Set that manages a set of pods
- Replica count can be adjusted as needed to scale the Replica Set out and back
- Replica Set successor to the ReplicationController object



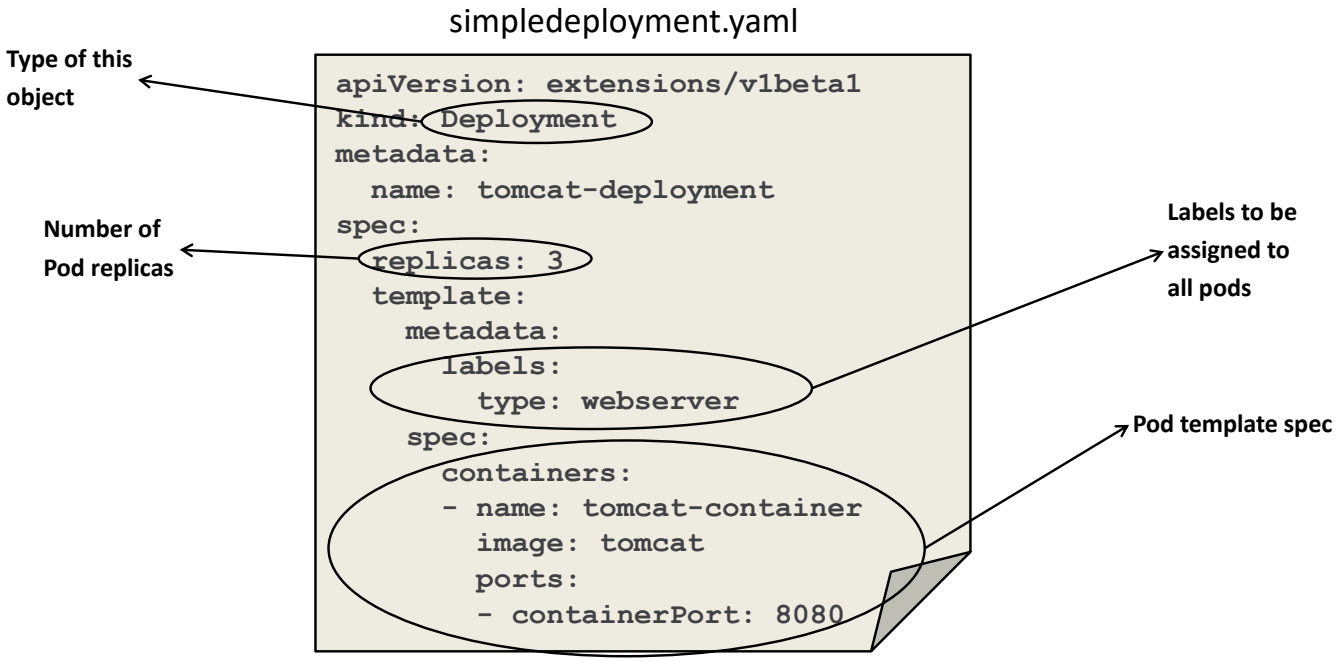
What is a Replica Set?

Provides scaling and high availability

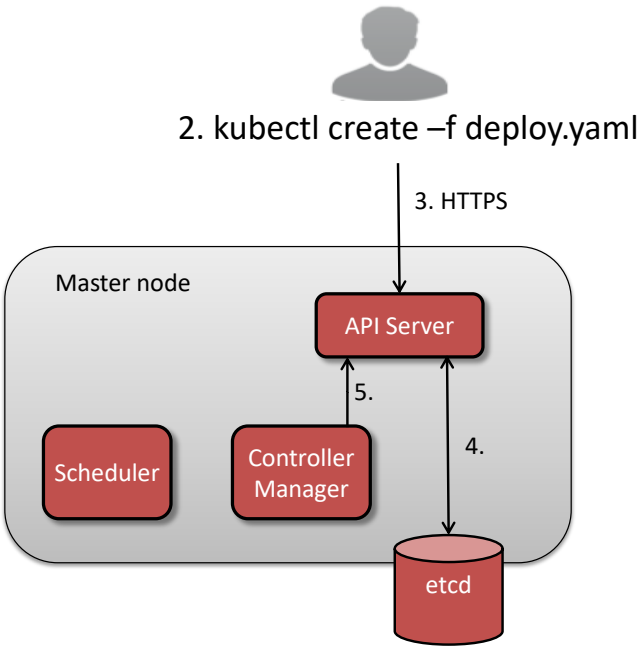
- Replica count can be changed to provide scaling on demand as needed
- If the node hosting a pod fails, the Kubernetes cluster will recreate the pod elsewhere to achieve the target number of replicas



Examining a Deployment Manifest File



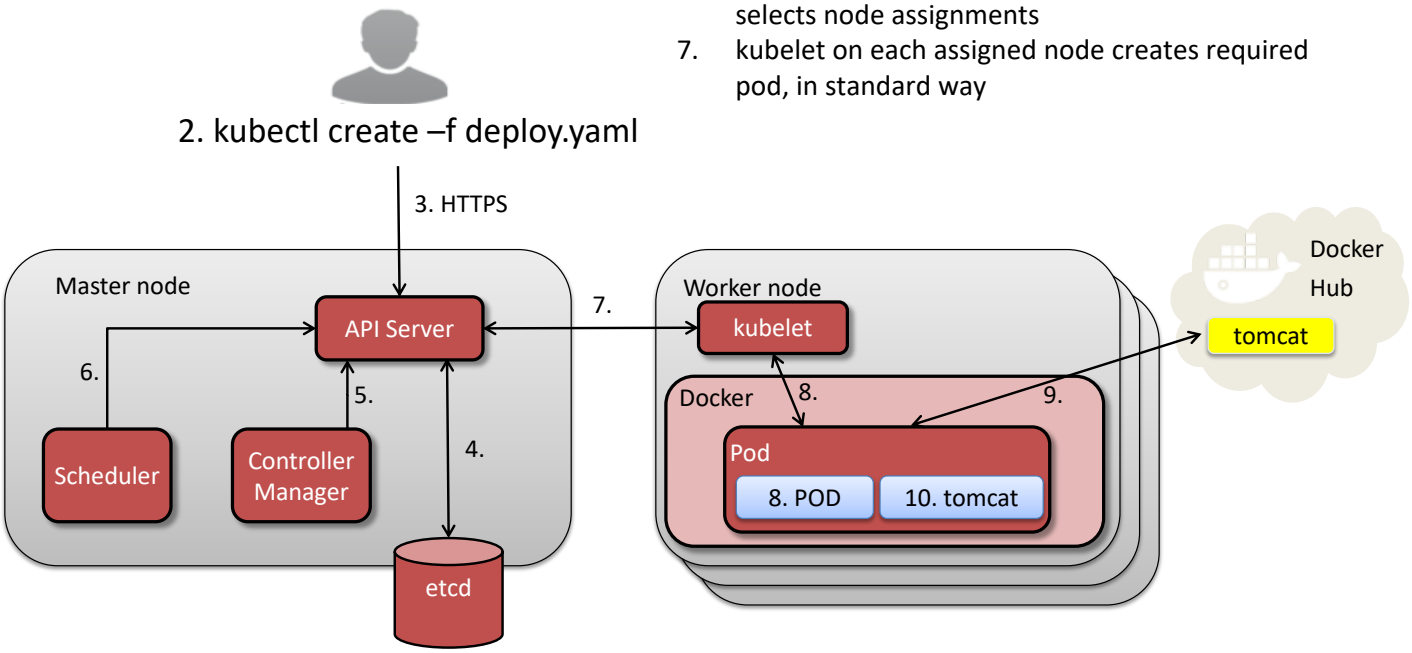
Deployment Creation Process



1. User writes a deployment manifest file
2. User requests creation of deployment from manifest via CLI
3. CLI tool marshals parameters into K8s RESTful API request (HTTP POST)
4. kube-apiserver creates new deployment object record in etcd, and new Replica Set object
5. kube-controller-manager sees new Replica Set and
   a. Evaluates state of existing vs. required replicas
   b. Submits pod creation requests to API to create required number of replicas

Deployment Creation Process

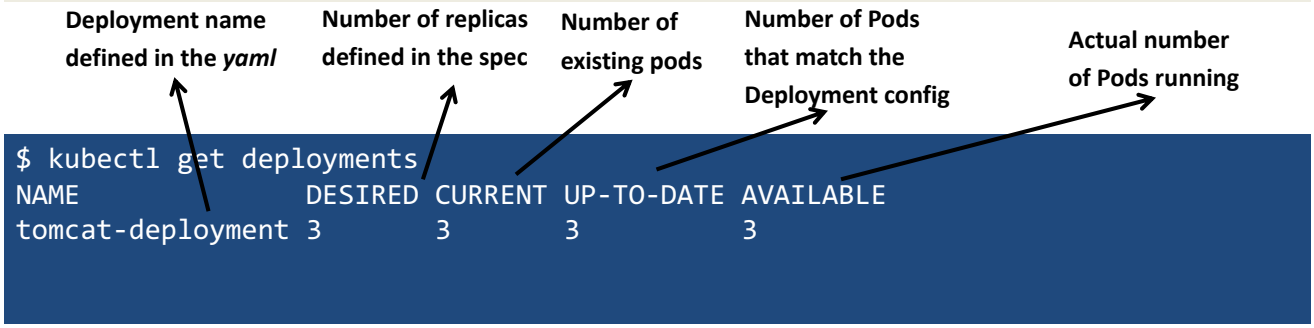
6. Scheduler sees new pod objects created and selects node assignments
7. kubelet on each assigned node creates required pod, in standard way



Deployments Management

Deployments Control ReplicaSets

The deployment creates a ReplicaSet that controls pod creation




```
$ kubectl get rs
NAME                          DESIRED CURRENT UP-TO-DATE AVAILABLE  AGE
tomcat-deployment-1699985759 3       3       3          3          2m
```

Replica Set Details

Replica Set name is <Deployment name>-<pod template hash value>

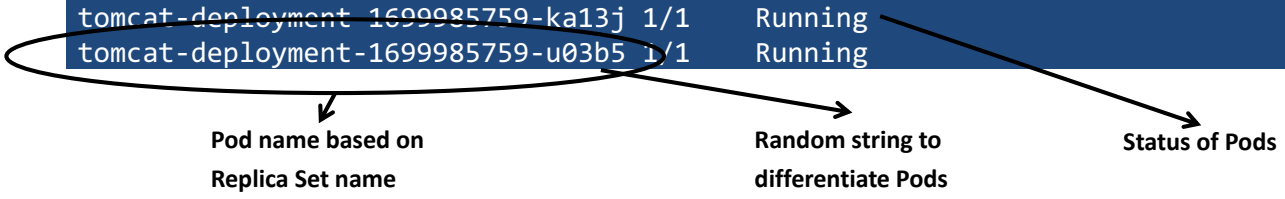
```
$ kubectl describe replicaset tomcat-deployment-1699985759
Name:          tomcat-deployment-1699985759
Namespace:     default
Image(s):      tomcat
Selector:      pod-template-hash=1699985759,type=webserver
Labels:        pod-template-hash=1699985759
               type=webserver
Replicas:      3 current / 3 desired
Pods Status:   3 Running / 0 Waiting / 0 Succeeded / 0 Failed
...
```

Selector uses labels from pod template and template hash

Pod Naming

Pod name is <Replica Set name>-<pod unique random string>

```
$ kubectl get pods
NAME                                READY   STATUS
tomcat-deployment-1699985759-h11sz 0/1     ContainerCreating
tomcat-deployment-1699985759-ka13j 0/1     ContainerCreating
tomcat-deployment-1699985759-u03b5 1/1     Running

$ kubectl get pods
NAME                                READY   STATUS
tomcat-deployment-1699985759-h11sz 1/1     Running
tomcat-deployment-1699985759-ka13j 1/1     Running
tomcat-deployment-1699985759-u03b5 1/1     Running
```



Modifying a Deployment to Trigger a Rollout

Multiple ways to change a Deployment configuration

- Change the manifest file and apply it via *kubectl apply*
- Change specific Deployment attribute via *kubectl set*
- Edit the Deployment config in the cluster via *kubectl edit*
- **Any changes to the Pod template will trigger a rollout of the Deployment to create and replicate new Pods with the new template**
  - **This means any changes – even things like pod labels!**

Pausing a Deployment

Pause a Deployment to temporarily halt rollout of updated pods

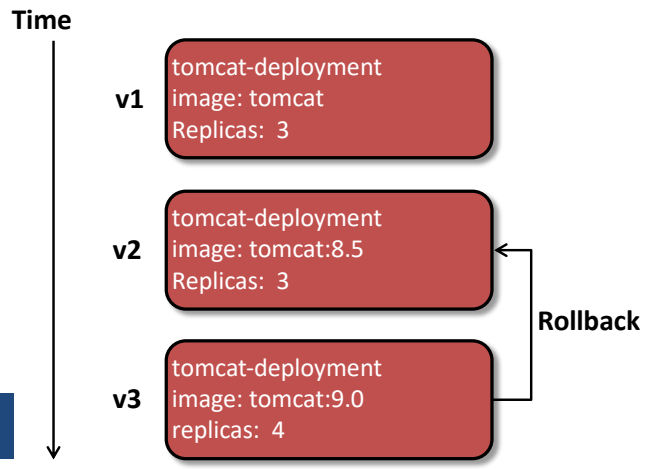
```
$ kubectl set image deployment/tomcat-deployment tomcat-container=tomcat:8.5.5;
kubectl rollout pause deployment/tomcat-deployment
deployment "tomcat-deployment" image updated
deployment "nginx-deployment" paused

$ kubectl rollout resume deployment/tomcat
deployment "tomcat-deployment" resumed

kubectl rollout status deployment/tomcat-deployment
deployment "tomcat-deployment" successfully rolled out
```

Checking Deployment Rollout History

Kubernetes tracks revisions made to a Deployment

- Users can query history of a deployment and see how many versions existed, and what change was made
  - Need to use *–record* flag on kubectl to record details of change commands
- History allows you to roll back state of a Deployment to a previous version

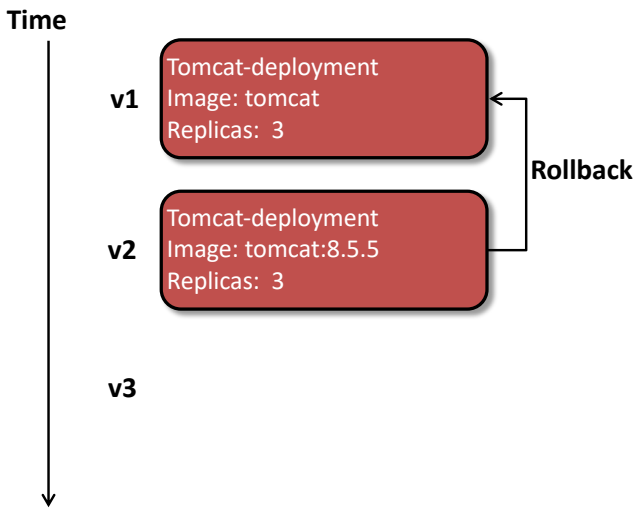
```
$ kubectl rollout history deploy/<deployment>
```



Rolling back a Deployment

Undoing Deployment changes via rollback operation

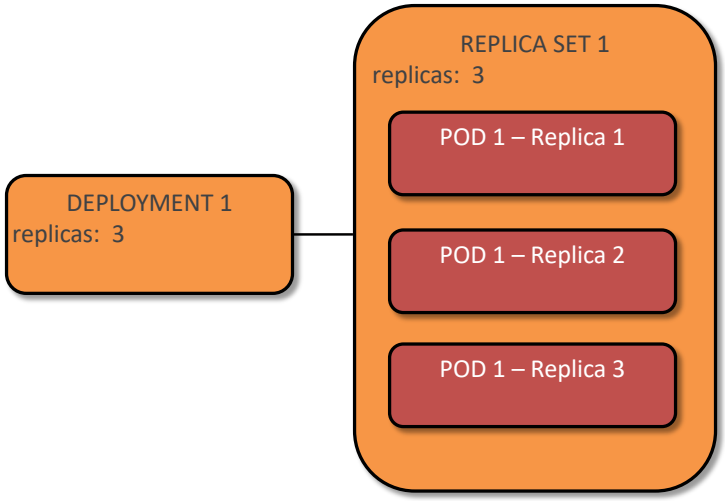
- You can undo the last change or roll back to a specific version
- The revision order will be changed to reflect the change
  - In this example, revision v1 will become the new revision v3
  - Version numbers increase monotonically
- Revision state stored in the corresponding Replica Sets



Deleting a Deployment

Deleting a Deployment will delete its Replica Set and all its Pods

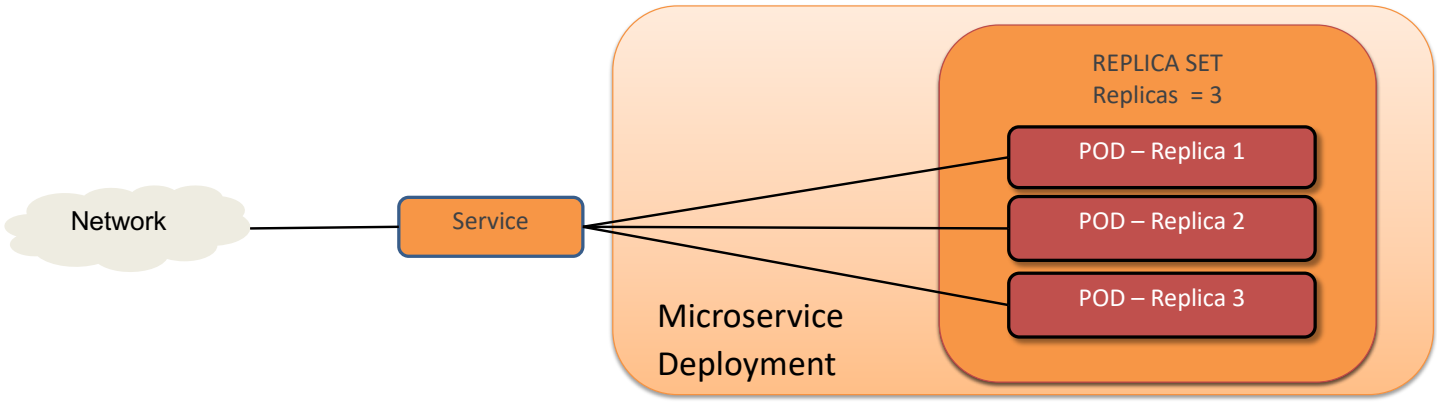
- By default, when Deployment is deleted, its Replica Sets are deleted
- Deletion of Replica Set cascades to deletion of pods managed by the rs
- Any Replica Sets reflecting previous versions of the Deployment will also be deleted



Scaling Deployments

Kubernetes Deployments for Microservices

- Applications built as services or collections of microservices easily modeled using Kubernetes Deployment and Service resources
- Key property is that application or component can be scaled horizontally



Basic Microservices Principles

- Application is composed of in(ter)dependent, independently deployable components
  - E.g. containerized components packaged as Docker images
- Design of application is service-oriented, with information exchange between components via standard interfaces, e.g.
  - REST/RPC APIs
  - Message Bus
- Maximize use of stateless components that can be scaled horizontally via replication
  - Horizontal scaling provides elastic capacity for handing demand

Application Scaling Strategies

Several ways to achieve scalability

- **Pre-defined application scale**: application deployed with a predefined, static number of resources => not the Kubernetes way
- **Manual scaling**: deployment scale reconfiguration driven by operator
- **Auto-scaling**: automatic scaling of resources based on a defined trigger (number of hits, CPU usage, etc)

Application Scaling Strategies

Manual scaling

- K8s supports manual scaling of Deployment to adjust replica count
- Operator just needs to adjust declaration of how many replicas are desired, and K8s system will manage to that number
  - Pod creation and placement completely automatic
  - Pods automatically re-created if nodes fail

```
$ kubectl scale --replicas=2 deployment/tomcat

$ kubectl scale --replicas=8 deployment/tomcat

$ kubectl scale --replicas=4 deployment/tomcat
```

```
$ kubectl edit deployment/tomcat
```

```
$ kubectl apply –f simpledeployment.yaml
```

Application Scaling Strategies

Auto-scaling in Kubernetes

- Kubernetes has a controller object for automatic scaling of Deployments (or Replica Sets or ReplicationControllers)
- HorizontalPodAutoscaler is control loop to adjust scale of pod set depending on one or more metrics, evaluated at configurable interval (default 30s)
  - Requires Heapster to be deployed on the cluster to provide metric data
  - Metrics include CPU and RAM utilization for pods
- Typical scenario: increase Deployment replica count when average Pod CPU utilization is above threshold value for specified period of time
- Note: application must support horizontal scaling!

Automatic Scaling of a Deployment

Creating a HorizontalPodAutoscaler resource

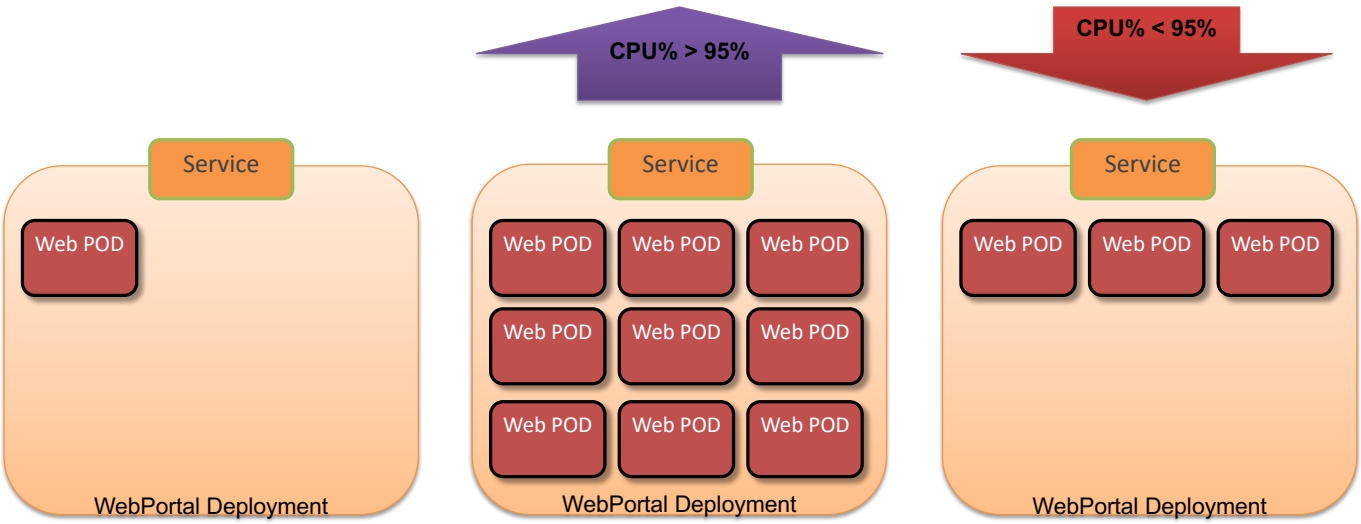
- The hpa is created similarly to any other Kubernetes resource
  - Use a manifest file for source tracking
  - Can create directly with *kubectl*
- 30 seconds is the default for considering the threshold
- Scaling is metrics-driven, either resource metrics from Heapster or custom metrics API

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: tomcat-autoscaler
spec:
  maxReplicas: 10
  minReplicas: 2
  scaleTargetRef:
    kind: Deployment
    name: tomcat-deployment
  targetCPUUtilizationPercentage: 75
```

simplescaler.yaml

Auto-Scaling of a Deployment

HorizontalPodAutoscaler with a 95% CPU usage threshold



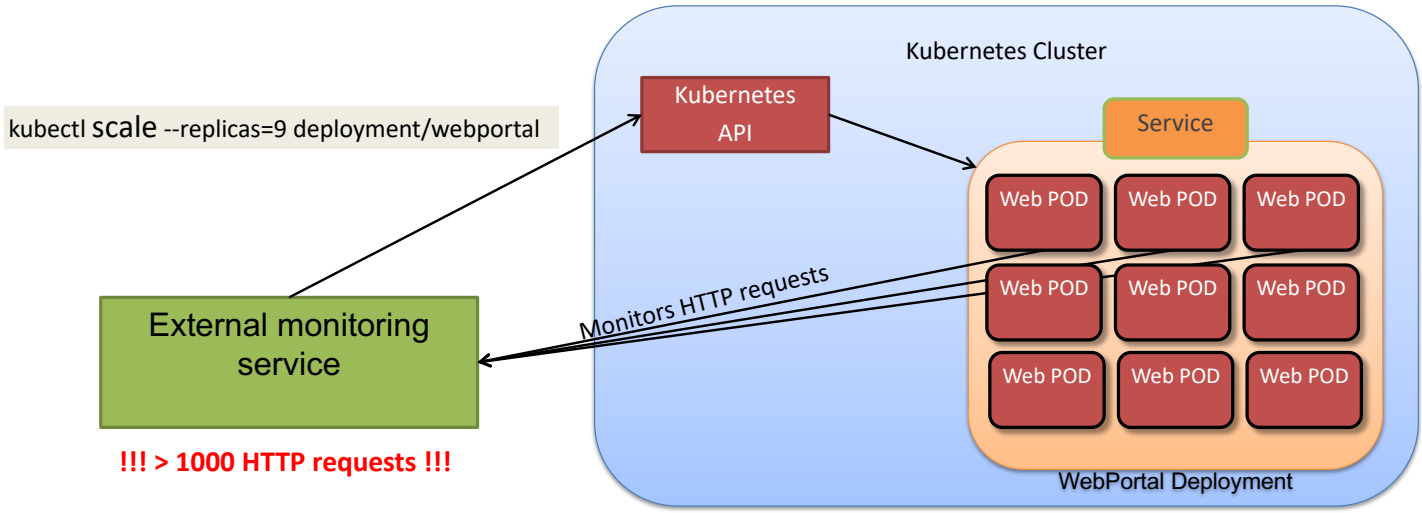
Application Scaling Strategies

Externally-driven auto-scaling

- Kubernetes API can be used to change Deployment replica counts remotely
- Any monitoring application that can send REST calls (e.g. Nagios) could be used to drive automatic scaling externally
- Very flexible approach, but more complex to implement
- **Needs robust testing before going into production**

Application Scaling Strategies

Auto-scaling driven by external system



Deployment Strategies Overview

What is a Deployment Strategy?

Approaches to manage risks on updating Deployments

- On each Deployment update/change, all pods in the deployment will be deleted and recreated
- Recreation process can have service impacts, especially for large Deployments
- A Deployment strategy defines how this rebuild process is done, to minimize downtime due to application failures or malfunctions

Types of Deployment Strategies

Kubernetes supports two basic strategies, but users can also leverage multiple Deployments when applying changes

- Strategies for single Deployments
  - Recreate
  - RollingUpdate
- Strategic approaches using two Deployments with a Service
  - Canary deployments
  - Blue/Green deployments

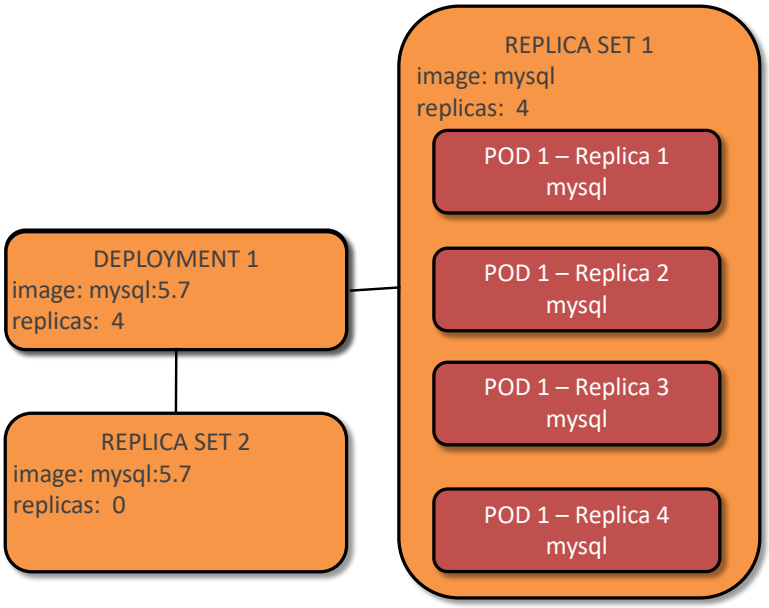
Each approach has a specific behavior and advantages/disadvantages.

Deployment Strategy: Recreate

Deployment Strategy: Recreate

Simplest strategy for deployments

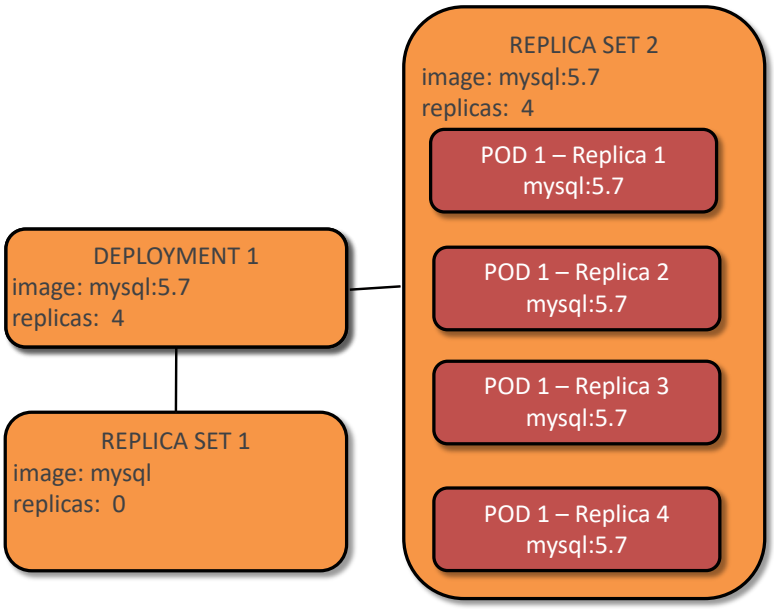
- When a change is made to a Deployment's spec, all Pods are removed and then recreated
  - Old Replica Set pods are killed
  - Then, new Replica Set starts pods
- May lead to downtime during the process while new pods are started



Deployment Strategy: Recreate

Simplest strategy for deployment updates

- When a change is made to a Deployment's template, all Pods are removed and then recreated
  - Old Replica Set pods are killed
  - New Replica Set starts pods
- May cause downtime due to delay between old pods terminating and new pods becoming available



Deployment Strategy: Recreate

Strategies are defined in the spec of a Deployment

- **strategy** parameter in Deployment spec sets the strategy to be used for updates
- If no parameter value is set, the default is **RollingUpdate**

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: tomcat-deployment
spec:
  replicas: 3
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        type: webserver
    spec:
      containers:
      - name: tomcat-container
        image: tomcat
        ports:
        - containerPort: 8080
```

Deployment Strategy: RollingUpdate

Deployment Strategy: RollingUpdate

RollingUpdate is DEFAULT strategy for Deployments

- When a change is made to the Deployment, the old Replica Set pods are scaled down as new pods are created by the new Replica Set
- A minimum number of running Pods is specified, so the Deployment will never be totally out of Pods to respond to service requests
- During the update process, the requested replica count may be temporarily exceeded

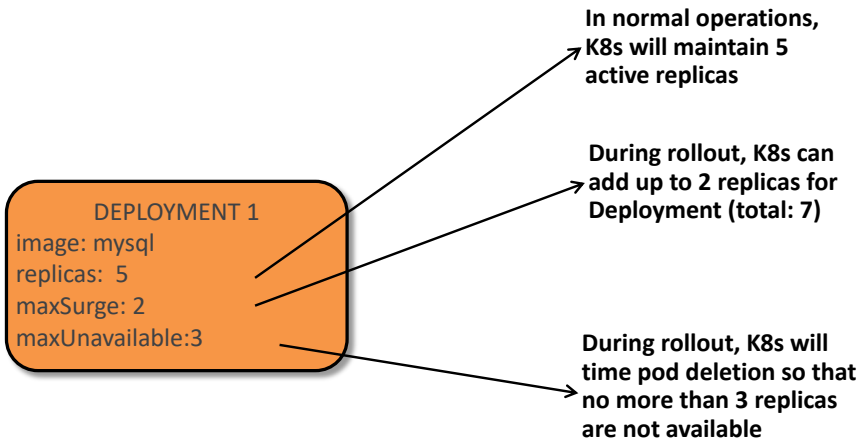
Deployment Strategy: RollingUpdate

Configure parameters to control the update process

```
...
metadata:
  name: tomcat-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 2
template:
    metadata:
      labels:
        type: webserver
...
```

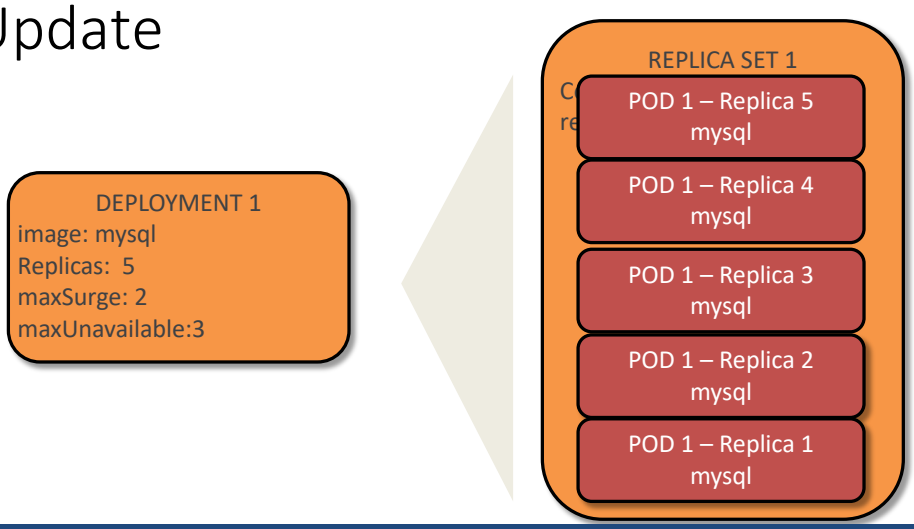
- **maxSurge**: number or percentage of additional Pods that can be created exceeding the replica count during update
  - Default value of 25%
- **maxUnavailable**: number of Pods that can be unavailable during the update
  - Default value of 25%

Deployment Strategy: RollingUpdate



Deployment Strategy: RollingUpdate

Initial State



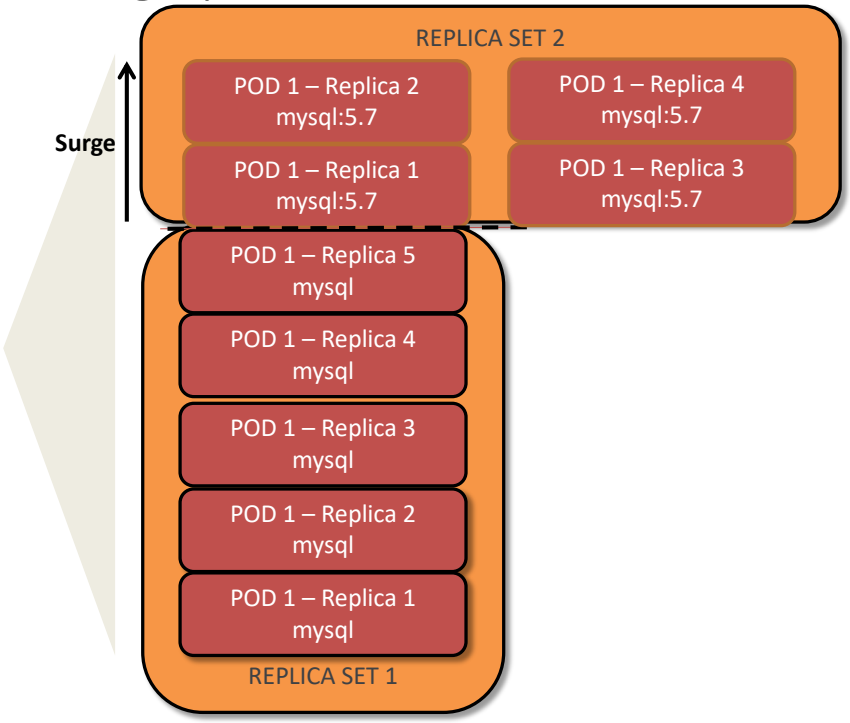

```
$ vi simpledeployment.yaml
...
    image: mysql:5.7
...
$ kubectl apply –f simpledeployment.yaml
```

Deployment Strategy: RollingUpdate

Rollout in progress

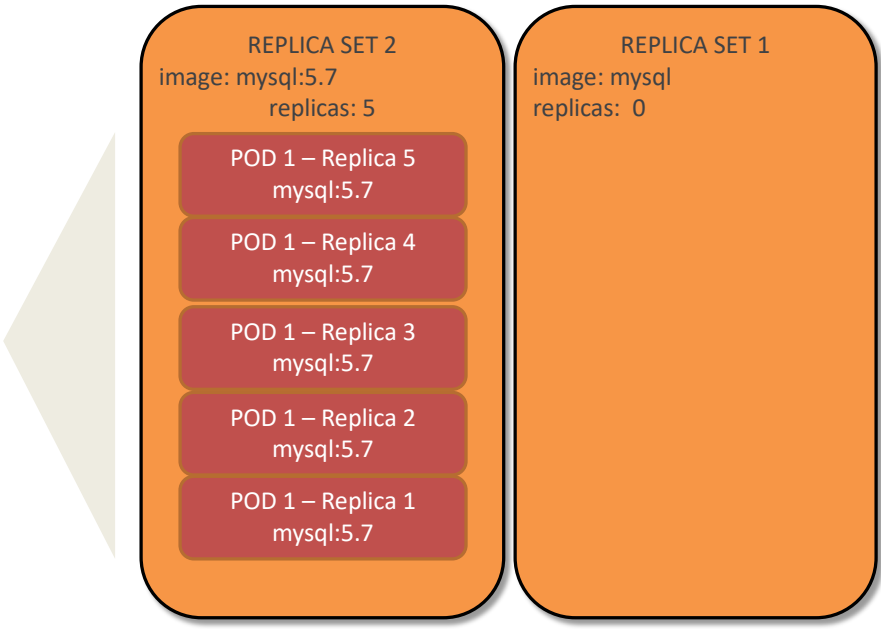
DEPLOYMENT 1
image: mysql:5.7
Replicas: 5
maxSurge: 2
maxUnavailable:3

- Initial surge of new pods on new Replica Set
- Original Replica Set scaled back as new RS scaled out



Deployment Strategy: RollingUpdate

Rollout complete



- By default, old, inactive Replica Set saved – previous version of the Deployment

Updating Using Multiple Deployments

RollingUpdate using Multiple Deployments

Controlled testing of new versions in production

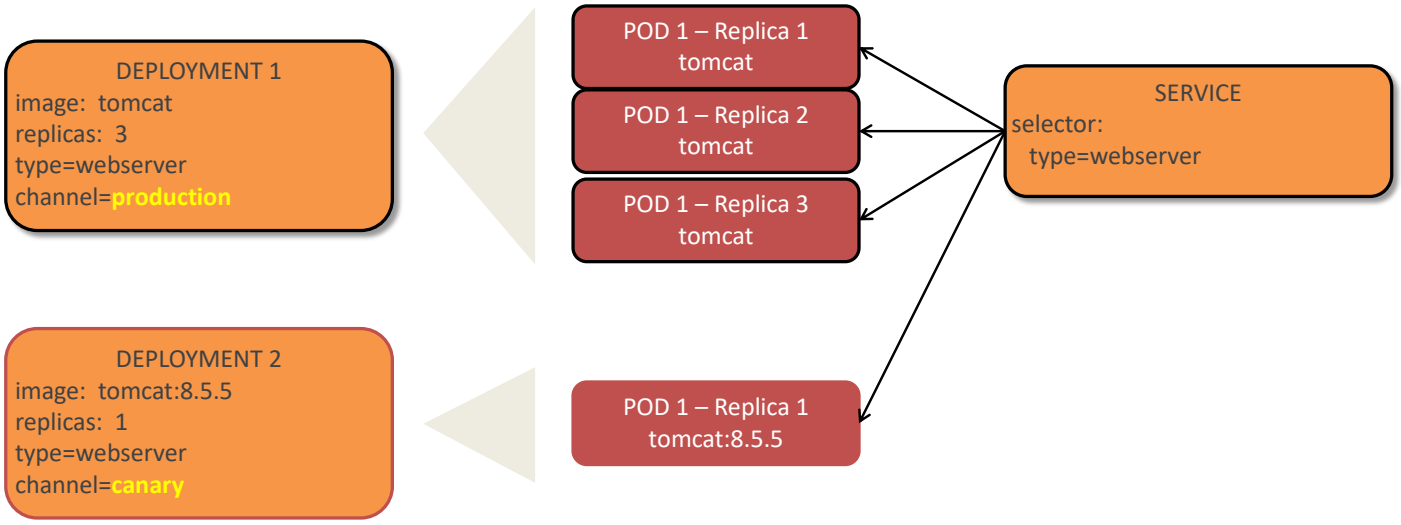
- Assume an application running as a Deployment, exposed as a Service
- To apply a new application version in production, a second Deployment can be used using labels in common with the first Deployment
  - Canary deployment allows for limited testing of new version in production
  - Blue/green deployment

Strategic Approach: Canary Deployment

Controlled testing of the update on production

- Consider a Service selecting pods from a Deployment of application pods
- In a canary deployment, a second Deployment (Canary Deployment) is created with pods for the new version, with labels matching the Service's selector
- Service directs some requests to pods on the Canary, allowing testing of changes in production
- If a malfunction is detected, it will only impact a small portion of the Pods and can be undone.

Strategic Approach: Canary Deployment



Strategic Approach: Canary Deployment

Decisions after running the canary Deployment in production

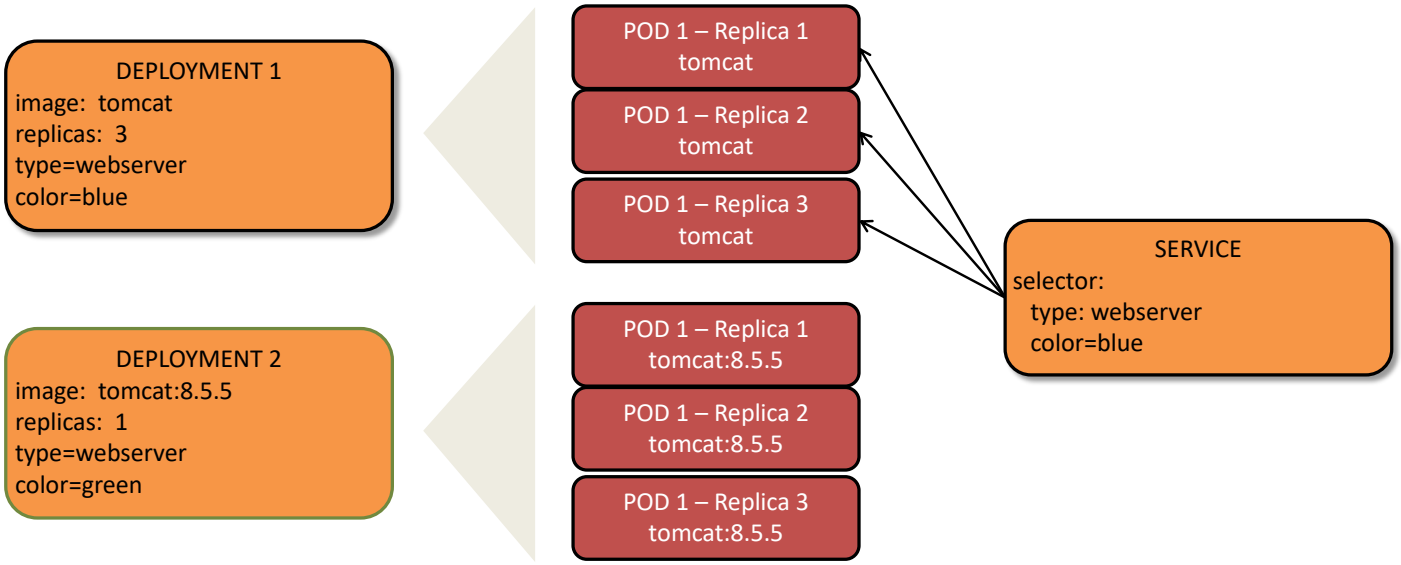
- If the application error rate is not increasing and Canary Deployment is stable:
  - The main Deployment can be updated to the newer version (using Rolling Update for example) and then the Canary can be discarded; OR
  - The Canary can be scaled up and reconfigured and the old Deployment can be discarded.
- If the test results in failure, the Canary deployment can be deleted

Strategic Approach: Blue/Green Deployment

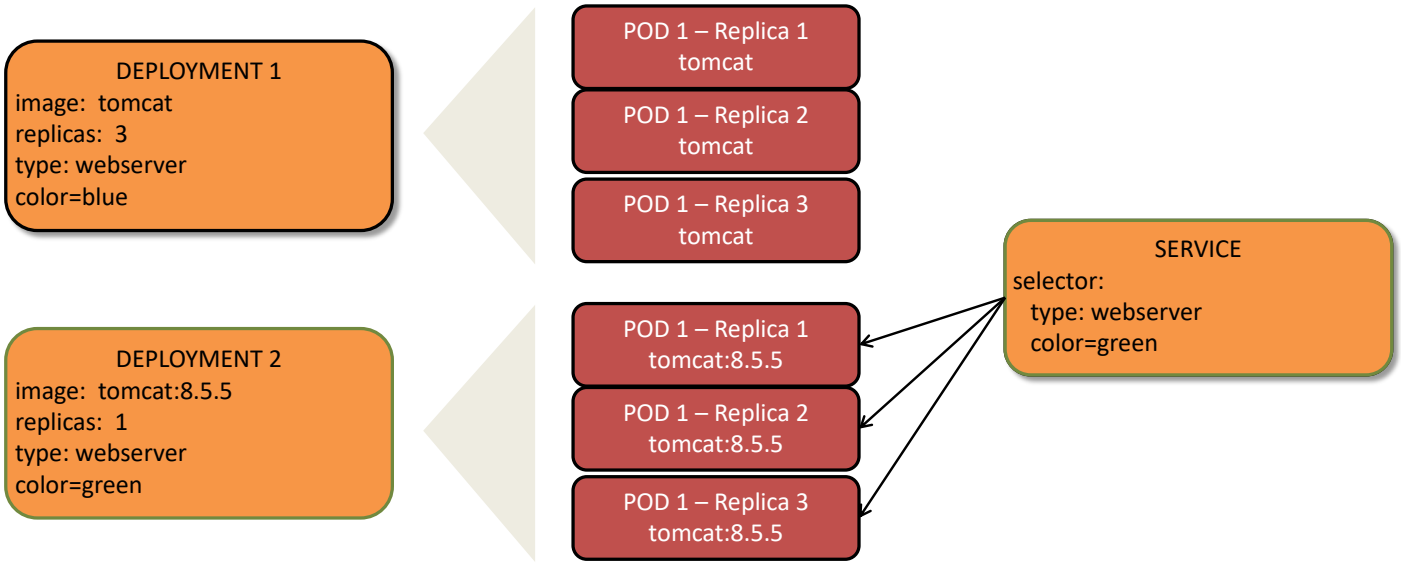
Complete environment switch from one version to another

- With a Blue/Green deployment, you create a new full-scale Deployment in addition to the current production Deployment
- Reconfiguring the pod label selector on the application's Service allows choice of directing requests to old Deployment or new Deployment
- Similar to effect of Replace strategy without application downtime

Strategic Approach: Blue/Green Deployment



Strategic Approach: Blue/Green Deployment



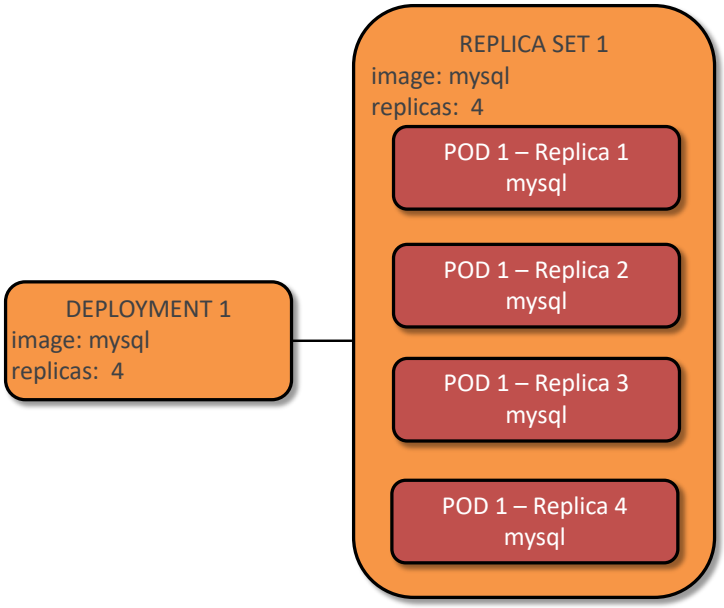
Module Summary

- Deployments overview
- Deployments management
- Scaling deployments
- Deployment strategies overview
- Deployment strategy: Recreate
- Deployment strategy: RollingUpdate
- Updating using multiple Deployments

Review: Kubernetes Deployments

- Deployments overview
- Deployments management
- Scaling deployments
- Deployment strategies overview

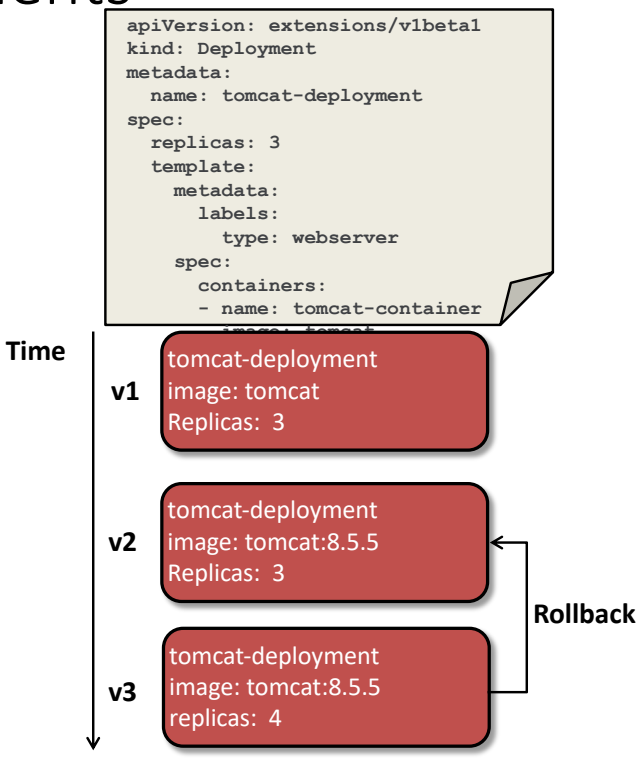
- Deployments allow you to declaratively manage pods, including replication
- Deployments support
  - Creating, rolling out, and rolling back changes to homogeneous set of pods
  - Scaling set of pods out and back



Review: Kubernetes Deployments

- Deployments overview
- Deployments management
- Scaling deployments
- Deployment strategies overview

- Deployments defined and managed like other K8s resources
- Deployments maintain version history and support rollout and rollback of changes



Review: Kubernetes Deployments

- Deployments overview
- Deployments management
- Scaling deployments
- Deployment strategies overview

- Deployments created with initial allocated scale
- Can adjust scale manually via API
- Use K8s HorizontalPodAutoscaler controller to automatically scale Deployment based on resource utilization

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: tomcat-autoscaler
spec:
  maxReplicas: 10
  minReplicas: 2
  scaleTargetRef:
    kind: Deployment
    name: tomcat-deployment
  targetCPUUtilizationPercentage: 75
```

Review: Kubernetes Deployments

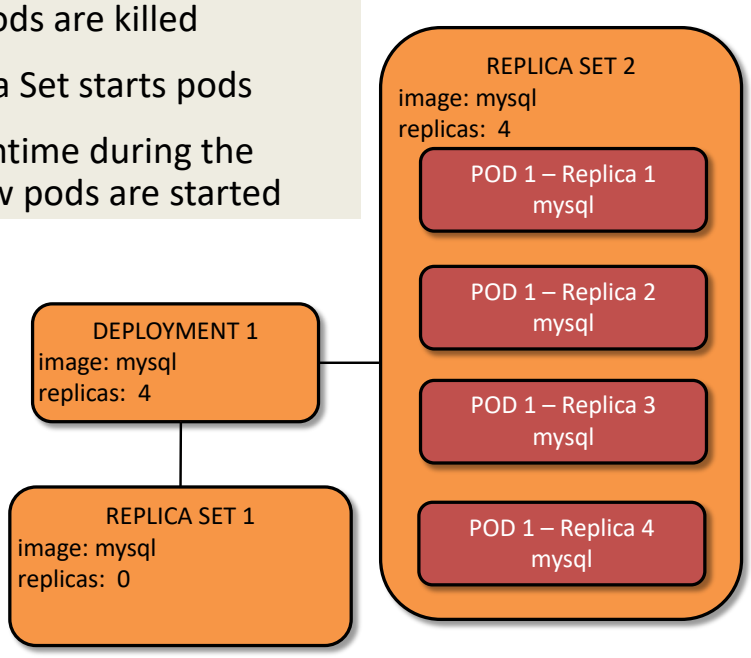
- Deployments overview
- Deployments management
- Scaling deployments
- Deployment strategies overview
- Deployment strategy: Recreate
- Deployment strategy: RollingUpdate
- Updating using multiple Deployments

- Two basic Deployment strategies
  - Replace
  - RollingUpdate (default)
- Additional strategies can use multiple Deployments with a Service

Review: Kubernetes Deployments

- Deployments overvie
- Deployments manag
- Scaling deployments
- Deployment strategi
- Deployment strategy: Recreate
- Deployment strategy: RollingUpdate
- Updating using multiple Deployments

- Old Replica Set pods are killed
- Then, new Replica Set starts pods
- May lead to downtime during the process while new pods are started



Review: Kubernetes Deployments

- Deployment strategies Overview
- Deployment strategy: Recreate
- Deployment strategy: RollingUpdate
- Updating using multiple Deployments

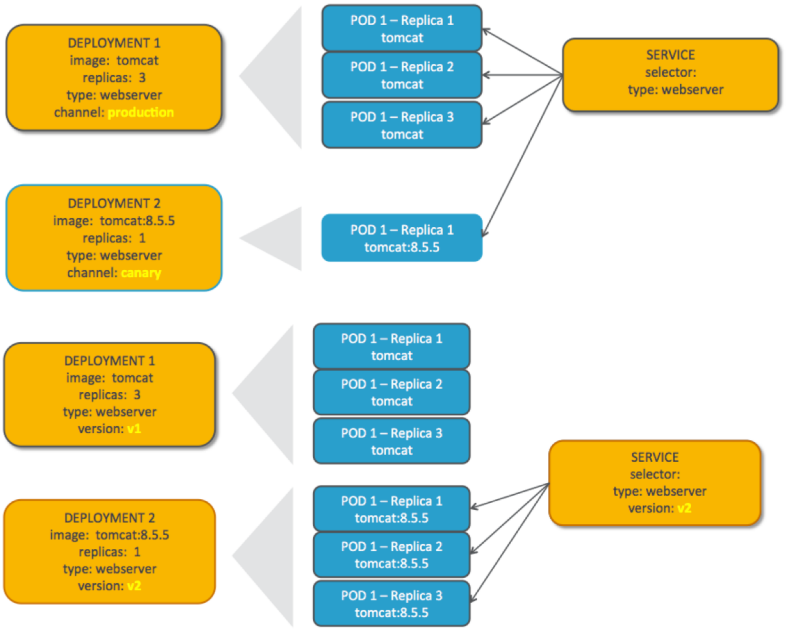
- Default strategy for Deployments
- New Replica Set pods scaled out while old RS scaled back to 0
- Can temporarily surge beyond replica count
- Can set minimum amount of pods to stay available during rollout

```
...
metadata:
  name: tomcat-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 2
template:
    metadata:
      labels:
        type: webserver
...
```

Review: Kubernetes Deployments

- Deployment strategies Overview
- Deployment strategy: Recreate
- Deployment strategy: RollingUpdate
- Updating using multiple Deployments

- Using two Deployments with a Service, can implement other strategic approaches
  - Canary deployment: new version in limited production
  - Blue/green deployment: rapid cutover from one version to another at scale



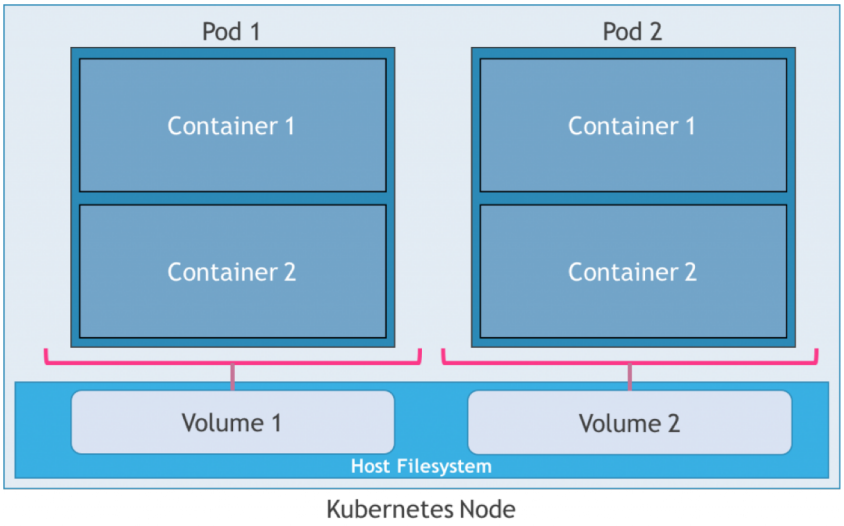
Lab: Deployments 1 & 2


Questions

Persistent Volumes

Host Based Storage

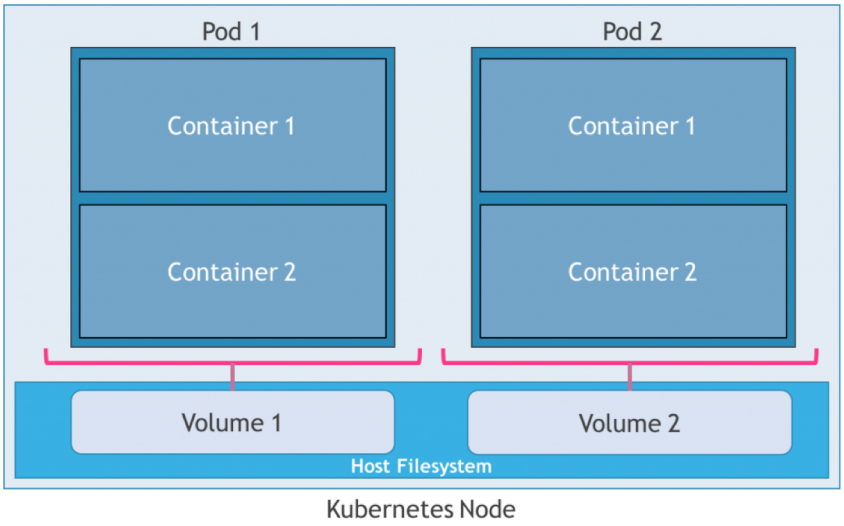
- Containers in a Pod share the same volume
- Host based storage
  - Local filesystem
- Removed when Pod is deleted
- Types of local storage
  - emptyDir
  - hostPath



Host Based Storage

- Common use cases
  - Scratch disk
  - Store temp config data

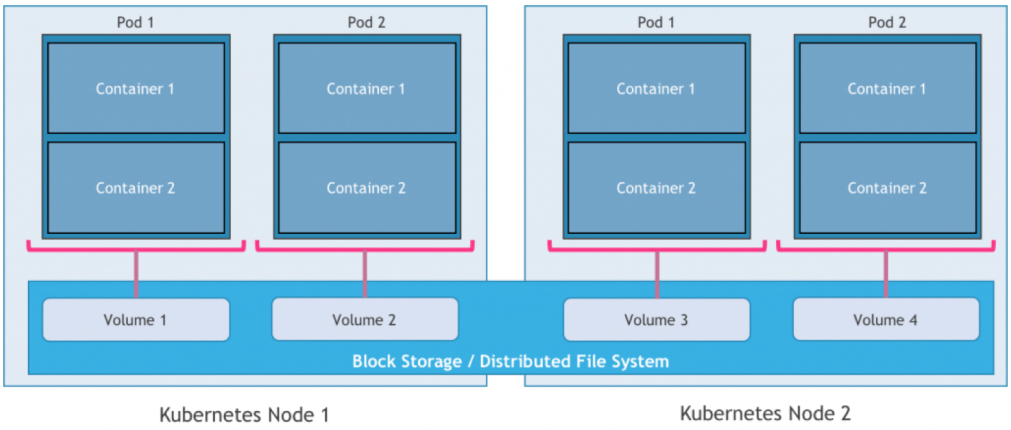
- hostPath
  - Directories on host
  - Owned by root



Non-Host Based Storage

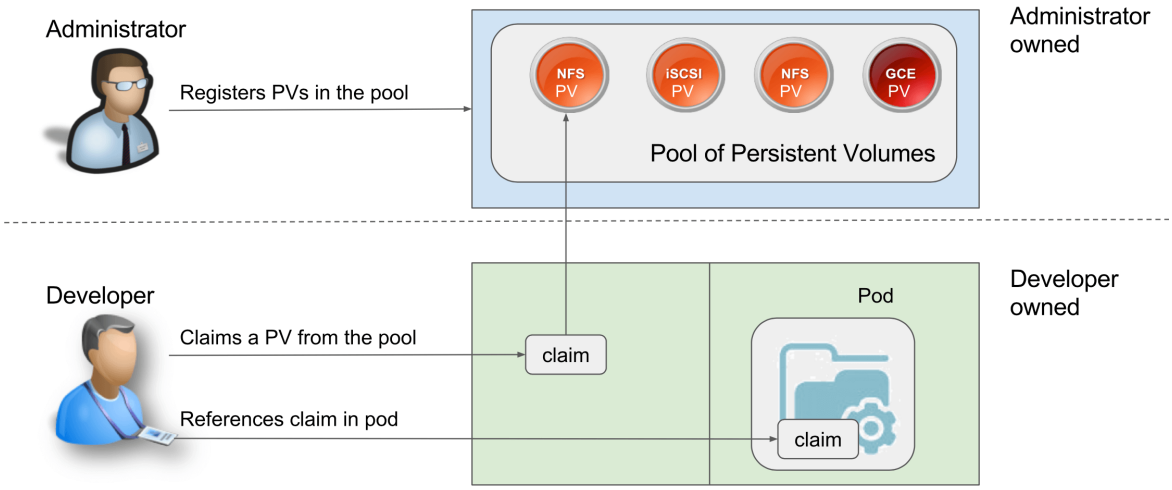
- Common use cases
  - Persistent data
  - DB, Stateful apps

- EBS
- GCE Persistent Disks
- NFS, NetApp, GlusterFS



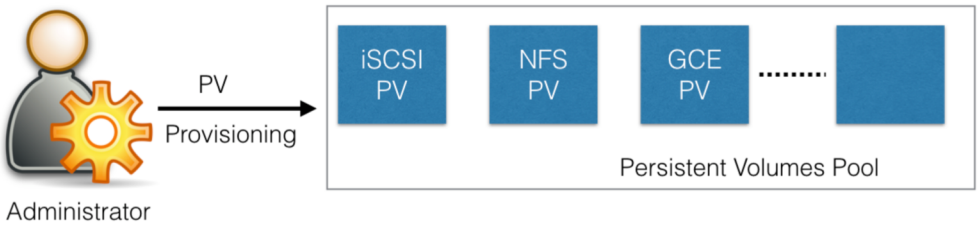
Persistent Volumes & Claims

- Operations creates PVs
  - Real storage details
  - Storage “LUN”
- Devs claim a PV
  - Self-service
  - Flexibility
  - Speed



Persistent Volumes

- Admin managed
- Quotas
- Access control
- Storage pools

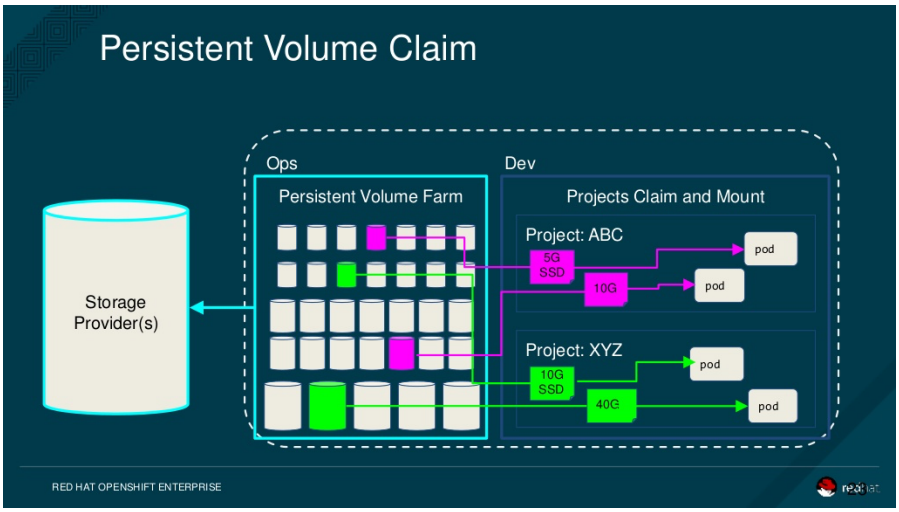


Persistent Volumes

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-aws
spec:
  capacity:
    storage: 12Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  awsElasticBlockStore:
    fsType: "ext4"
    volumeID: "vol-f37a03aa"
```

Persistent Volume Claims

- Allows Devs to claim storage without worrying about management.
- Devs only see storage assigned to projects they are in.



Persistent Volume Claims

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pv-claim
  labels:
    app: mongodb
spec:
   accessModes:
    - ReadWriteOnce
   resources:
      requests:
         storage: 10Gi
```

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: rsvp-db
spec:
  replicas: 1
  template:
    metadata:
      labels:
        appdb: rsvpdb
    spec:
      containers:
      - name: rsvpd-db
        image: mongo:3.3
        ports:
        - containerPort: 27017
        volumeMounts:
        - name : mongodb-persistent-storage
          mountPath : /data/db
      volumes:
      - name: mongodb-persistent-storage
        persistentVolumeClaim:
          claimName: mongodb-pv-claim
```


Questions

ConfigMap

ConfigMap

ConfigMap

- Many applications require configuration via:
  - Config Files
  - Command-Line Arguments
  - Environment Variables
- These need to be decoupled from images to keep portable
- ConfigMap API provides mechanisms to inject containers with configuration data
- Store individual properties or entire config files/JSON blobs
- Key-Value Pairs

ConfigMap

- Not meant for sensitive information
- PODs or controllers can use ConfigMaps

1. Populate the value of environment variables
2. Set command-line arguments in a container
3. Populate config files in a volume

```
kind: ConfigMap
apiVersion: v1
metadata:
   creationTimestamp: 2016-02-18T19:14:38Z
   name: example-config
   namespace: default
data:
   example.property.1: hello
   example.property.2: world
   example.property.file: |-
      property.1=value-1
      property.2=value-2
      property.3=value-3
```

ConfigMap from directory

- 2 files in docs/user-guide/configmap/kubectl
  - game.properties
  - ui.properties

game.properties

```
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

ui.properties

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

ConfigMap from directory

```
kubectl create configmap game-config --from-file=docs/user-guide/configmap/kubectl
```

```
kubectl describe configmaps game-config
Name: game-config
Namespace: default
Labels: <none>
Annotations: <none>
Data
====
game.properties: 121 bytes
ui.properties: 83 bytes
```

ConfigMap from directory

```
kubectl get configmaps game-config -o yaml
```

```
apiVersion: v1
data:
    game.properties: |-
        enemies=aliens
        lives=3
        …
ui.properties: |
       color.good=purple
       …
kind: ConfigMap
metadata:
    creationTimestamp: 2016-02-18T18:34:05Z
    name: game-config
    namespace: default
    resourceVersion: "407"-
    selfLink: /api/v1/namespaces/default/configmaps/game-config
    uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

ConfigMap from files

```
kubectl create configmap \
game-config-2 \
--from-file=file1 \
--from-file=file2
```

```
kubectl get configmaps \
game-config-2 \
-o yaml
```

```
apiVersion: v1
data:
    game.properties: |-
       enemies=aliens
       lives=3
       …
ui.properties: |
      color.good=purple
      …
kind: ConfigMap
metadata:
    creationTimestamp: 2016-02-18T18:52:05Z
    name: game-config-2
    namespace: default
    resourceVersion: ”516"-
    selfLink: /api/v1/namespaces/default/configmaps/game-config-2
    uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

ConfigMap options

--from-file=/path/to/directory
--from-file=/path/to/file1 (/path/to/file2)
Literal key=value: --from-literal=special.how=very

ConfigMap in PODs

- Populate Environment Variables

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: special-config
    namespace: default
data:
    special.how: very
    special.type: charm
```

OUTPUT

```
SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm
```

```
apiVersion: v1
kind: Pod
metadata:
    name: dapi-test-pod
spec:
  containers:
      - name: test-container
        image: busybox
        command: ["/bin/sh", "-c, "env" ]
        env:
          - name: SPECIAL_LEVEL_KEY
            valueFrom:
              configMapKeyRef:
                name: special-config
                key: special.how
  restartPolicy: Never
```

ConfigMap Restrictions

- ConfigMaps must be created before they are consumed
- ConfigMaps can only be referenced by objects in the same namespace
- Quota for ConfigMap size not implemented yet
- Can only use ConfigMap for PODs created through API server.

Lab: ConfigMap

Secrets

Application Secrets

What secrets do applications have?

- Database credentials
- API credentials & endpoints (Twitter, Facebook etc.)
- Infrastructure API credentials (Google, Azure, AWS)
- Private keys (TLS, SSH)
- Many more!

Application Secrets

It is a bad idea to include these secrets in your code.

- Accidentally push up to GitHub with your code
- Push into your file storage and forget about
- Etc.

Application Secrets

There are bots crawling GitHub searching for secrets

Real life example:
Dev put keys out on GitHub, woke up next morning with a ton of emails and missed calls from Amazon

- 140 instances running under Dev's account.
- $2,375 worth of Bitcoin mining

Create Secret

- Designed to hold all kinds of sensitive information
- Can be used by Pods (filesystem & environment variables) and the underlying kubelet when pulling images

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
Type: Opaque
data:
   password: mmyWfoidfIuL==
   username: NyhdOKwB
```

Pod Secret

```
kind: Pod
metadata:
  name: secret-env-pod
spec:
 containers:
 - name: mycontainer
   image: redis
   env:
   - name: SECRET_USERNAME
       valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
```

Volume Secret

```
spec:
  containers
  - name: mycontainer
    image: redis
    volumeMounts:
    - name: “secrets”
      mountPath: “/etc/my-secrets”
      readOnly: true
  volumes:
   - name: “secrets”
      secret:
        secretName: “mysecret”
```

Lab: Secrets

Role Based Access Controls

RBAC

- Considered stable in Kubernetes 1.8 and later
- To enable start apiserver with
  - --authorization-mode=RBAC
  - This allows for more fine-grained control of access permissions

RBAC Roles

- Role used to grant access to resources in a single namespace

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
```

RBAC ClusterRole

- ClusterRole used to grant access to resources that are cluster specific, not namespace specific

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

RBAC RoleBinding

- Grant permissions defined in a role to user or set of users

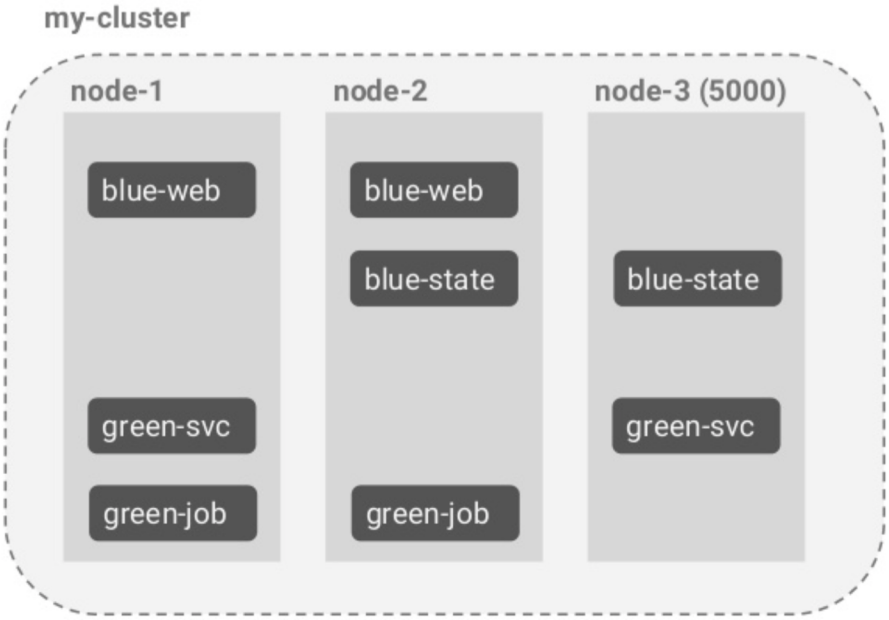
```
# This role binding allows "jane" to read pods in the "default" namespace.
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
   kind: Role
   name: pod-reader
   apiGroup: rbac.authorization.k8s.io
```

Without fine-grained Access:

- Authorization at cluster level
- All pods have same authorization

Without controlled scheduling:

- Lack flexibility for multi-workload

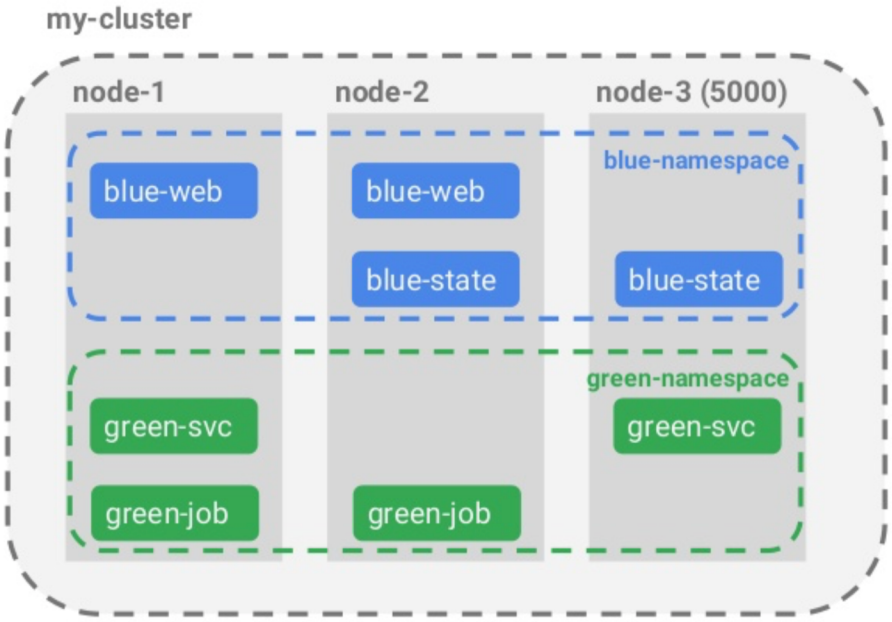


Introducing RBAC:

- Per-namespace/ resource, role, action

Examples:

- **Alice** can list **Eng** services, but not **HR**
- **Bob** can create Pods in **Test** namespace, but not in **Prod**
- **Scheduler** can read **Pods** but not **Secrets**




THANK
YOU