REPLICATED

Supporting Kubernetes Applications in End Customer Environments

There are many guides with cheat sheets for quickly getting up to speed on Kubernetes and kubectl basics. In contrast, this guide is specifically for teams who build, deliver, and support Kubernetes products deployed into **customer environments**. When delivering into a Kubernetes environment that you don't control, there are several things to be aware of, and some workflows and patterns that we've found invaluable for accelerating remediation in the field.

If you're familiar with managing/supporting **non**-Kubernetes applications in customer environments, but your engineering team is migrating your application(s) to Kubernetes, then this guide is for you.

Workloads

Workloads manage Pods. Pods represent running instances of an application. The Kubernetes API uses control loops to ensure that an application is running and healthy. When a user deploys an application to Kubernetes, the *apiserver* figures out which Nodes are eligible for new assignments, and then it *schedules* workloads to run on those Nodes.

Workload types

Pods	The basic unit of work. A Pod is a group of containers that share an IP address that is assigned to run on a specific node in the cluster. Each other concept in the "Workloads" section represents a higher level abstraction that represents a specific desired state of one or more Pods.
ReplicaSets	A ReplicaSet specifies a group of Pods. The desired state of a ReplicaSet is "N number of Pods based on a specific template, running at all times". If a node fails, any Pods controlled by a ReplicaSet will be *rescheduled* to healthy nodes to meet that desired state.
Deployments	Deployments are the typical workload Controller. A Deployment schedules Pods to any node in the cluster by creating and destroying ReplicaSets. Most useful for stateless applications, but can be used with stateful applications as well.
StatefulSets	The StatefulSets Controller schedules a specified number of Pods to worker Nodes, but makes the guarantee that once a Pod has been scheduled to a node, that Pod will always get rescheduled to the same node and will attach to the same storage volume. Most useful for stateful components like databases, queues and caches.
Jobs & CronJobs	The Jobs controller creates Pods that are intended to run once and then exit without restarting. Most useful for running one-off processes, such as database migrations. The CronJobs controller creates Jobs that are executed at a given time, described by the traditional `cron` notation. The CronJob controller will wait for the right time, then execute a Pod to perform some work.
DaemonSets	The DaemonSets Controller will schedule one Pod per node across all nodes in the cluster. Most useful for applications that would typically run as daemons in a traditional Linux VM, such as collectd or syslog.

Interacting with workloads

`kubectl explain deployments`	View the documentation for the Deployment manifest.
`kubectl get --all-namespaces deploy,statefulset,daemonset, jobs,cronjobs`	View all the workloads running across all namespaces in the cluster.
`kubectl get po` `kubectl get pods`	List the Pods in the *default* namespace.
`kubectl get po $pod -o yaml`	Print the manifest for a Pod in YAML format.
`kubectl get po $pod -o json`	Print the manifest for a Pod in JSON format.
`kubectl get po -n $ns -o wide`	List all the Pods in the $ns namespace and show their IP addresses and the node they're running on.
`kubectl run -ti -rm busybox --image=busybox -- sh`	Launch a busybox Pod which will be deleted when you exit. Remove the "--rm" for it to persist.
`kubectl exec -it -n rook-ceph rook-ceph-tools -- bash`	Launch a shell in an existing deployment's Pod, in this case rook-ceph-tools within the rook-ceph namespace.
`kubectl delete all -l $LABEL=$VALUE`	Delete all services, deployments, Pods, jobs, and other workloads with a matching Label. Will not delete secrets, configmaps and some other resources.
`kubectl logs -f $pod`	Continuously print the logs from $pod in the *default* namespace.
`kubectl logs $pod -c $ctr --previous`	Dump the logs from a previous version of a given container in a given Pod that has multiple containers.
`kubectl rollout status $deployment` `kubectl rollout history $deployment`	Show the status or history of the changes made to a Deployment. If --record was used, print the reason for the change.
`kubectl scale deployment redis --replicas=3`	Scale the Deployment "redis" to 3 replicas.
`kubectl get events --sort-by= .metadata.creationTimestamp`	List Events sorted by timestamp.

Services

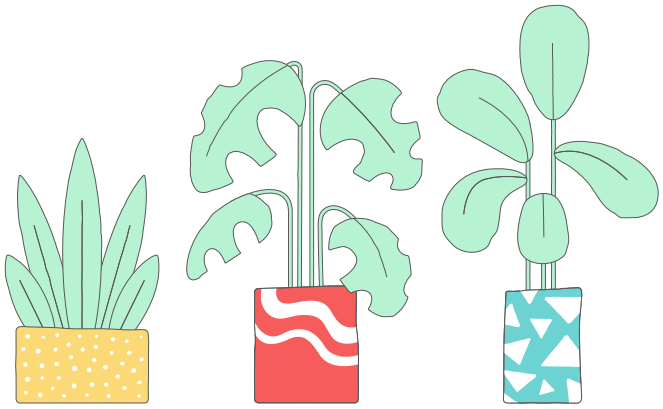
A Service describes a software-defined load balancer for Pods. Since Pods are constantly being replaced, Pod IPs are not constant. A Service provides an easy way to address a group of Pods. Services use a *selector* to specify the Pods that they will operate on. When an application makes a request to a Service, the Service will load balance the request to any Pods with a *label* matching the Service's *selector*.

Service Types

ClusterIP	Creates a virtual IP to forward in-cluster traffic to Pods matching the Service's selector.
NodePort	Open a port on each node (usually in the 30000-32768 range) that will forward traffic from outside the cluster to the Service's matching Pods.
LoadBalancer	Create a cloud-based load balancer like an AWS ELB or GCP Forwarding Rule for the service. Cloud Provider Only -- requires the kube-controller-manager process to be configured to manage cloud resources. If a LoadBalancer Service is stuck in the Pending state, the cluster may not be configured to manage cloud resources.

Handy Service Commands

`kubectl explain service`	View the documentation for the Service manifest.
`kubectl get service -n $NAMESPACE` `kubectl get svc -n $NAMESPACE`	List the Services in a given namespace and show ClusterIP and any listening ports.
`kubectl get svc -o wide`	List the Services in the *default* namespace and show the *selector* for each.
`kubectl get endpoints` `kubectl get ep`	Show health/ready endpoints for each service in a namespace.
`kubectl expose deployment httpd --port=80 --target-port=80`	Create a Service in the *default* namespace that exposes a Deployment "httpd" on port 80 within the cluster.
`kubectl port-forward svc/$MYAPP 8800:8800`	Forward traffic from localhost:8800 to the service $MYAPP.



Nodes

A node is typically a virtual machine or bare metal server that runs Pods. Pods are a logical group of containers. A node shares resources like CPU, memory, etc., with the Pods it runs. A node communicates with the Kubernetes API Server using a process called kubelet. Kubelet talks to the container orchestrator to spin up the containers that represent a Pod. A Node understands which Pods it is supposed to be running and communicates those Pods' the status (including success/failure) back to the API Server. You can use *kubectl describe node* or *kubectl get node -o json* to inspect the status of a node.

Interacting with Nodes

`kubectl drain $node`	Drain a given node in preparation for maintenance -- removes all Pods and "cordons" the node, preventing new Pods from being scheduled to it.
`kubectl uncordon $node`	Post-maintenance, mark a given node as schedulable.
`kubectl top node $node`	Show metrics for a given node.
`kubectl describe node $node`	Show node information and events.
`kubectl get node $node -o json`	Show raw node json -- useful for inspecting status fields (see to the right).

Useful Node Status Fields

Addresses	DNS and IP addresses known to be attributed to this node.
Allocatable, Capacity	Shows the available / total resources on the node, including CPUs, Storage, memory, and more.
Conditions	A collection of events that describe the node. For example a healthy node will have a *KernelDeadlock* status of *False* with a message "kernel has no deadlock". There are many such conditions which can help an operator understand the health of a node.
Images	A list of docker images already present on the node.
NodeInfo	Basic info like Architecture, Kernel Version, OS Image, and Kubernetes Component versions.

Storage

Persistent storage in Kubernetes is handled by PersistentVolume and PersistentVolumeClaim objects. PersistentVolumes represent attached block storage. PersistentVolumeClaims represent a relationship between Persistent Volumes and the Pods that consume them. PersistentVolumes can either be created dynamically (typically with an automated call to the cloud provider), or statically (where volumes are created by an administrator and then made available to Kubernetes as a resource). Users or ServiceAccounts can submit a *claim* for some storage. If a free PersistentVolume matches a Pod's *claim*, then the PersistentVolume will be assigned to that Pod. PersistentVolume definitions are static and cannot be changed once created.

Storage Objects Types

PersistentVolume	A static definition for some kind of persistent storage. May be a local filesystem path, or may be part of an IaaS system such as AWS EBS/EFS or Azure File Storage. Cluster Scoped
PersistentVolumeClaim	A request for some storage to be assigned to a Pod. Namespace Scoped
StorageClass	A generalized specification for a volume. Used to dynamically provision PVs without administrator intervention. Used in combination with a *provisioner*. Cluster Scoped

Inspecting Storage Objects

`kubectl get --all-namespaces pv,pvc`	Describe all the PVCs in all namespaces and list the PVs that are in the cluster scope.
`kubectl describe storageclass default`	Show the specification for the *default* StorageClass.
`kubectl describe pvc -n $ns $pvc`	Show the status of a given PVC in a given namespace. Note the Status which will show Bound if the PVC has a matching & bound PV, and the Events table which may describe any errors that occurred when trying to mount the PV. See the PVC's capacity, read/write mode, and StorageClass.

Kubernetes RBAC

RBAC overview

Roles, RoleBindings, ServiceAccounts

Kubernetes RBAC is a way of managing Authorization - *who* can perform *which* actions on the Kubernetes API. ServiceAccounts are non-human users - typically used by applications inside Kubernetes that need to talk to the Kubernetes API. Permissions describe what actions can be performed. A Permission is an association of a verb (list, create, update, etc.) with the name of a Kubernetes object (Pods, Services, etc.). A Role is a set of permissions. RoleBindings associate a Role to a user, group, or service account.

Subject	A Subject refers to the actor (a user or ServiceAccount) that performs some action on the cluster.
Role	A Role defines the actions that a Subject may perform on a specific set of objects in a specific namespace.
RoleBinding	A RoleBinding assigns a Role to a list of Subjects.
ClusterRole	A ClusterRole is a set of actions that may be performed on objects across the entire cluster.
ClusterRoleBinding	A ClusterRoleBinding assigns a Role or ClusterRole to a Subject across all namespaces.

`kubectl get -n $namespace role,rolebinding,serviceaccount`	List the Roles, RoleBindings, and ServiceAccounts in a given namespace.
`kubectl get clusterrole, clusterrolebinding`	List the ClusterRoles and ClusterRoleBindings.
`kubectl describe rolebinding $role-binding`	Show the Role and Subject that are governed by a given RoleBinding.
`kubectl auth can-i --list`	Show all authorizations my current user can perform.

TROUBLESHOOT

Does all of this sound hard? Replicated built troubleshoot.sh to automate common support tasks and expedite remediation for trustless troubleshooting in customer-controlled environments. For more info, book a demo today.

Embedded Trouble Analyzers

Identify common application-level errors for faster diagnosis of application issues for trustless troubleshooting with custom redaction from support bundles to remove sensitive data before sharing.

Preflight Checks

Ensure predictable deployments, with advanced validation of the customer environment for the requirements of your application.

Troubleshoot Collectors

Collect customer data with redaction of sensitive information for secure support and troubleshooting.

Configuration Validation

Test the validity of the configuration parameters entered by your customer post-install.

`kubectl krew install preflight support bundle`	Install Troubleshoot.sh plugins with krew.dev
`kubectl preflight spec.yaml`	Run preflight checks based on a local yaml file, running collectors and analyzers to surface pass/warn/fail insights about the cluster.
`kubectl preflight https://preflight.replicated.com`	Use a URL-hosted spec instead of a local file.
`kubectl support-bundle spec.yaml`	Run Collectors and Show analysis based on a local SupportBundle spec, leaving behind a .tar.gz with all collected information for easy sharing. As with preflight, can also be pointed at a remote URL.
`kubectl support-bundle https://help.example.com/support-bundle.yaml --redactors=./redact.yaml`	Run a support bundle based on a remote spec, with custom redaction logic applied.

Example Specs

Preflight

Check Node count and size

```
apiVersion: troubleshoot.sh/v1beta2
kind: Preflight
metadata:
  name: preflight-tutorial
spec:
  analyzers:

    - nodeResources:
        checkName: One node must have 16
GB RAM and 8 CPU Cores
        filters:
          allocatableMemory: 16Gi
          cpuCapacity: “8”
        outcomes:
          - fail:
              when: count() < 1
              message: Cannot find a node
with sufficient memory and cpu
          - pass:
              message: Sufficient CPU and
memory is available
```

Redactor

Remove sensitive usernames from collected logs

```
apiVersion: troubleshoot.sh/v1beta2
kind: Redactor
metadata:
  name: my-redactor-name
spec:
  redactors:
  - name: remove passwords
    removals:
      regex:
      - redactor: (another)(?P<mask>.*)
(here)
      - selector: ‘S3_ENDPOINT’
        redactor: ‘(“value”: “).*(“)’
      yamlPath:
      - “abc.xyz.*”
```

SupportBundle

Collect Cluster Info and logs from Pods matching app=api

```
apiVersion: troubleshoot.sh/v1beta2
kind: SupportBundle
metadata:
  name: my-application-name
spec:
  collectors:
    - clusterInfo: {}
    - clusterResources: {}
    - logs:
        selector:
          - app=api
        namespace: default
```