EQUANT kubectl Cheat Sheet

We'll cover all important Kubernetes commands to communicate with cluster control panes. You can use this kubectl/ kubernetes cheat sheet however you'd like. It's a great resource to prepare for interviews with, or easy reference to breeze through application projects.

Here is a link to kubectl cheat sheet pdf that you can download and access it offline whenever and wherever needed.

Kubectl Autocomplete

- **BASH**

To set up an autocomplete in bash within the current shell, you must install the bash-completion package using the following command.

```
source <(kubectl completion bash)
```

For adding the autocomplete permanently to your bash shell, run the following command.

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

You can use a short name for Kubectl, as shown below.

```
alias k=kubectl
complete -F __start_kubectl k
```

- **ZSH**

To set up an autocomplete in zsh within the current shell, you need to run the following command.

```
source <(kubectl completion zsh)
```

You can also run the following command to make it permanent.

```
echo "[[ $commands[kubectl] ]] && source <(kubectl completion zsh)" >>
~/.zshrc
```

Kubectl Context and Configuration

The next set of commands will cover display, context, and user information.

Display merged kubeconfig settings with this command:

```
kubectl config view
```

Use several kubeconfig files simultaneously and view the merged config with this command:

```
KUBECONFIG=~/.kube/config:~/.kube/kubconfig2
kubectl config view
```

Get the password of the e2e user with this command:

```
kubectl config view -o jsonpath='{.users[?(@.name ==
"e2e")].user.password}'
```

Run this command to display the first user:

```
kubectl config view -o jsonpath='{.users[].name}'
```

Display the entire list of the users with this command:

```
kubectl config view -o jsonpath='{.users[*].name}'
```

Display the list of context:

```
kubectl config get-contexts
```

Display the current context:

```
kubectl config current-context
```

Set the default context to the my-cluster-name:

```
kubectl config use-context my-cluster-name
```

Add a new user to your kubeconf supporting basic authentication:

```
kubectl config set-credentials kubeuser/foo.kubernetes.com
--username=kubeuser --password=kubepassword
```

Save the namespace permanently for all subsequent kubectl commands in that context:

```
kubectl config set-context --current --namespace=ggckad-s2
```

Set a context with a specific username and namespace:

```
kubectl config set-context gce --user=cluster-admin --namespace=foo \
  && kubectl config use-context gce
```

Delete the user named foo:

```
kubectl config unset users.foo
```

Set or show context/namespace with a short alias:

```
alias kx='f() { [ "$1" ] && kubectl config use-context $1 || kubectl config
current-context ; } ; f'
alias kn='f() { [ "$1" ] && kubectl config set-context --current
--namespace $1 || kubectl config view --minify | grep namespace | cut -d" "
-f6 ; } ; f'
```

Kubectl Apply

The “apply” helps manage the applications through files that define Kubernetes resources. You can easily create and update resources within the cluster by running kubectl apply. It is one of the commonly practiced ways for managing applications on production.

Creating Objects

Objects in Kubernetes are persistent entities that represent the state of your cluster.

Specifically, they can describe:

- Containerized applications running (and on which nodes)
- Available resources to those applications
- Policies about application behavior, like upgrades and restart policies

A Kubernetes object is a "record of intent." After you create the object, the Kubernetes system will constantly work to ensure that the object exists. By creating an object, you're telling the Kubernetes system about how your cluster's workload will look (desired state).

If you want to work with Kubernetes objects (create, modify, or delete), you need to use the Kubernetes API. The kubectl command-line interface (CLI) makes it easier for you to make the necessary Kubernentes API calls. Interested in using the Kubernetes API directly? Try out one of the client libraries.

You can define the Kubernetes manifest file in YAML or JSON. This manifest file comes with the extension as .yaml, .yml, and .json, etc.

The following are the commands that allow you to workaround the manifest file. “Apply” is used to push the required changes based on your configuration files.

Create resources:

```
kubectl apply -f ./my-manifest.yaml
```

Create from multiple files:

```
kubectl apply -f ./my1.yaml -f ./my2.yaml
```

Create resources in all manifest files in dir:

```
kubectl apply -f ./dir
```

**** the “create” command will help in generating the new resources from files or standard input devices.**

Create resources from url:

```
kubectl apply -f https://git.io/vPieo
```

Start a single instance of Nginx using the following command:

```
kubectl create deployment nginx --image=nginx
```

Create a job that will print "Hello World:"

```
kubectl create job hello --image=busybox:1.28 -- echo "Hello World"
```

Create a CronJob that will print "Hello World" every minute:

```
kubectl create cronjob hello --image=busybox:1.28   --schedule="*/1 * * *
*" -- echo "Hello World"
```

Retrieve the documentation for pod manifests:

```
kubectl explain pods
```

Create multiple YAML objects from stdin:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: busybox-sleep
spec:
  containers:
  - name: busybox
    image: busybox:1.28
    args:
    - sleep
    - "1000000"
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox-sleep-less
spec:
  containers:
  - name: busybox
    image: busybox:1.28
    args:
    - sleep
    - "1000"
EOF
```

Create a secret with several keys:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: $(echo -n "s33msi4" | base64 -w0)
  username: $(echo -n "jane" | base64 -w0)
EOF
```

Viewing and Finding Resources

In Kubernetes, we can define a resource as an endpoint present in the Kubernetes API. It is responsible for storing the collection of specific sorts of API objects. For example, the resource named built-in pods is a collection of pod objects.

We can use a custom resource to extend the Kubernetes API. It is not necessary that a custom resource should always be available in the default installation of Kubernetes. Moreover, custom resources have made it possible for us to create various core Kubernetes functions, which make Kubernetes more modular in nature.

Dynamic registration makes it possible for custom resources to appear or disappear in a running cluster. Also, cluster admins have the right to update custom resources without concerning the cluster itself. Like the built-in resources, users can also create and access the objects of custom resources when they are installed.

“Get” Commands with Basic Outputs

These commands will help you fetch cluster data from various sources:

- **kubectl get services:** # for displaying all services in the namespace
- **kubectl get pods --all-namespaces:** # for displaying all pods in all namespaces
- **kubectl get pods -o wide:** # for displaying all pods in the current namespace, with more details
- **kubectl get deployment my-dep:** # for displaying a particular deployment
- **kubectl get pods:** # for displaying all pods in the namespace
- **kubectl get pod my-pod -o yaml:** # to get a pod's YAML

“Describe” Commands to Display the Verbose Output

```
kubectl describe nodes my-node
```

```
kubectl describe pods my-pod
```

Commands for Listing, Period Trees, and More

List services sorted by name:

```
kubectl get services --sort-by=.metadata.name
```

List pods sorted by restart count:

```
kubectl get pods --sort-by='.status.containerStatuses[0].restartCount'
```

List PersistentVolumes sorted by capacity:

kubectl get pv --sort-by=.spec.capacity.storage

Obtain the version label of all pods with label app=cassandra:

```
kubectl get pods --selector=app=cassandra -o \
  jsonpath='{.items[*].metadata.labels.version}'
```

Get the value of a key with dots, e.g. 'ca.crt:'

```
kubectl get configmap myconfig \
  -o jsonpath='{.data.ca\.crt}'
```

View all worker nodes:

For this command, use a selector for excluding the results that have a label # named 'node-role.kubernetes.io/master.'

```
kubectl get node --selector='!node-role.kubernetes.io/master'
```

Get all running pods in the namespace:

```
kubectl get pods --field-selector=status.phase=Running
```

Get the ExternalIPs of all nodes:

```
kubectl get nodes -o
jsonpath='{.items[*].status.addresses[?(@.type=="ExternalIP")].address}'
```

List pod names of Pods that belong to Particular RC "jq:"

This command is useful for transformations that are too complex for jsonpath, and can be found here.

```
sel=${$(kubectl get rc my-rc --output=json | jq -j '.spec.selector | to_entries
| .[] | "\(.key)=\(.value),"')%?}
echo $(kubectl get pods --selector=$sel
--output=jsonpath={.items..metadata.name})
```

Show all pod labels:

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

Check which nodes are ready:

```
JSONPATH='{range .items[*]}{@.metadata.name}:{range
@.status.conditions[*]}{@.type}={@.status};{end}{end}' \
 && kubectl get nodes -o jsonpath="$JSONPATH" | grep "Ready=True"
```

Get the output decoded secrets without external tools:

```
kubectl get secret my-secret -o go-template='{{range $k,$v := .data}}{{"###
"}}{{$k}}{{"\n"}}{{$v|base64decode}}{{"\n\n"}}{{end}}'
```

List all secrets currently in use by a pod:

```
kubectl get pods -o json | jq
'.items[].spec.containers[].env[]?.valueFrom.secretKeyRef.name' | grep -v
null | sort | uniq
```

List all containerIDs of initContainer of all pods:

This command is helpful when cleaning up stopped containers, while avoiding removal of initContainers.

```
kubectl get pods --all-namespaces -o jsonpath='{range
.items[*].status.initContainerStatuses[*]}{.containerID}{"\n"}{end}' | cut
-d/ -f3
```

List events sorted by timestamp:

```
kubectl get events --sort-by=.metadata.creationTimestamp
```

Compare current and future cluster states:

```
kubectl diff -f ./my-manifest.yaml
```

Create a period-delimited tree of all keys returned for nodes:

This command is helpful when locating a key within a complex nested JSON structure.

```
kubectl get nodes -o json | jq -c 'paths|join(".")'
```

Create a period-delimited tree of all keys returned for pods, etc.:

```
kubectl get pods -o json | jq -c 'paths|join(".")'
```

Create an ENV for all pods:

This command assumes you have a default container for the pods, default namespace and that the `env` command is supported. It is helpful when running any supported command across all pods, not just `env.`

```
for pod in $(kubectl get po --output=jsonpath={.items..metadata.name}); do
echo $pod && kubectl exec -it $pod -- env; done
```

Updating Resources

- Roll the update "www" containers of "frontend" deployment, updating the image:

```
kubectl set image deployment/frontend www=image:v2
```

- Check the history of deployments, including the revision:

```
kubectl rollout history deployment/frontend
```

- Rollback to the previous deployment:

```
kubectl rollout undo deployment/frontend
```

- Rollback to a specific revision:

```
kubectl rollout undo deployment/frontend --to-revision=2
```

- Watch the rolling update status of "frontend" deployment until completion:

```
kubectl rollout status -w deployment/frontend
```

- Roll the restart of the "frontend" deployment:

```
kubectl rollout restart deployment/frontend
```

- Force replace, delete and then re-create the resource:

This command may cause a service outage.

```
kubectl replace --force -f ./pod.json
```

- Create a service for a replicated nginx, which serves on port 80 and connects to the containers on port 8000:

```
kubectl expose rc nginx --port=80 --target-port=8000
```

- Update a single-container pod's image version (tag) to v4:

```
kubectl get pod mypod -o yaml | sed 's/\(image: myimage\):.*$/\1:v4/' |
kubectl replace -f -
```

- Add a label:

```
kubectl label pods my-pod new-label=awesome
```

- Add an annotation:

```
kubectl annotate pods my-pod icon-url=http://goo.gl/XXBTWq
```

- Autoscale a deployment named “foo:”

```
kubectl autoscale deployment foo --min=2 --max=10
```

Patching Resources

These commands will patch resources as required.

- Partially update a node:

```
kubectl patch node k8s-node-1 -p '{"spec":{"unschedulable":true}}'
```

- Update a container's image:

Keep in mind that the spec.containers[*].name is required because it's a merge key

```
kubectl patch pod valid-pod -p
'{"spec":{"containers":[{"name":"kubernetes-serve-hostname","image":"new
image"}]}}'
```

- Update a container's image using a json patch with positional arrays:

```
kubectl patch pod valid-pod --type='json' -p='[{"op": "replace", "path":
"/spec/containers/0/image", "value":"new image"}]'
```

- Deploy livenessProbe using a json patch with positional arrays:

```
kubectl patch deployment valid-deployment  --type json   -p='[{"op":
"remove", "path": "/spec/template/spec/containers/0/livenessProbe"}]'
```

- Add a new element to a positional array:

```
kubectl patch sa default --type='json' -p='[{"op": "add", "path":
"/secrets/1", "value": {"name": "whatever" } }]'
```

Editing Resources

These commands will edit resources as required.

- Edit the service named “docker-registry:”

```
kubectl edit svc/docker-registry
```

- Use an alternative editor:

```
KUBE_EDITOR="nano" kubectl edit svc/docker-registry
```

Scaling Resources

These commands will scale resources as required.

- Scale replicaset named 'foo' to 3:

```
kubectl scale --replicas=3 rs/foo
```

- Scale a resource specified in "foo.yaml" to 3:

```
kubectl scale --replicas=3 -f foo.yaml
```

- Scale mysql to 3 (when the deployment named mysql's current size is 2):

```
kubectl scale --current-replicas=2 --replicas=3 deployment/mysql
```

- Scale multiple replication controllers:

```
kubectl scale --replicas=5 rc/foo rc/bar rc/baz
```

Deleting Resources

The following commands are used to delete the resources as required.

- Delete a pod using the type and name specified in pod.json:

```
kubectl delete -f ./pod.json
```

- Delete a pod with no grace period:

```
kubectl delete pod unwanted -now
```

- Delete pods and services with the same names "baz" and "foo:"

```
kubectl delete pod, service baz foo
```

- Delete pods and services with label name=myLabel:

```
kubectl delete pods, services -l name=myLabel
```

- Delete all pods and services in namespace my-ns:

```
kubectl -n my-ns delete pod, svc --all
```

- Delete all pods matching the awk pattern1 or pattern2:

```
kubectl get pods  -n mynamespace --no-headers=true | awk
'/pattern1|pattern2/{print $1}' | xargs  kubectl delete -n mynamespace pod
```

Interacting with Running Pods

This next set of commands will show you how to interact with running pods:

- Dump pod logs (stdout):

```
kubectl logs my-pod
```

- Dump pod logs, with label name=myLabel (stdout):

```
kubectl logs -l name=myLabel
```

- Dump pod logs (stdout) for a previous instantiation of a container:

```
kubectl logs my-pod --previous
```

- Dump pod container logs (stdout, multi-container case):

```
kubectl logs my-pod -c my-container
```

- Dump pod logs, with label name=myLabel (stdout):

```
kubectl logs -l name=myLabel -c my-container
```

- Dump pod container logs (stdout, multi-container case) for a previous instantiation of a container:

```
kubectl logs my-pod -c my-container --previous
```

- Stream pod logs (stdout):

```
kubectl logs -f my-pod
```

- Stream pod container logs (stdout, multi-container case):

```
kubectl logs -f my-pod -c my-container
```

- Stream all pods logs with label name=myLabel (stdout):

```
kubectl logs -f -l name=myLabel --all-containers
```

- Run a pod as interactive shell:

```
kubectl run -i --tty busybox --image=busybox:1.28 -- sh
```

- Start a single instance of nginx pod in the namespace of mynamespace:

```
kubectl run nginx --image=nginx -n mynamespace
```

- Run a pod nginx and write its spec into a file called pod.yaml:

```
kubectl run nginx --image=nginx
--dry-run=client -o yaml > pod.yaml
```

- Attach pod to Running Container:

```
kubectl attach my-pod -i
```

- Listen on port 5000 on the local machine and forward to port 6000 on my-pod:

```
kubectl port-forward my-pod 5000:6000
```

- Run a command in existing pod (1 container case):

```
kubectl exec my-pod -- ls /
```

- Access interactive shell to a running pod (1 container case)

```
kubectl exec --stdin --tty my-pod -- /bin/sh
```

- Run a command in existing pod (multi-container case):

```
kubectl exec my-pod -c my-container -- ls /
```

- Display metrics for a given pod and its containers:

```
kubectl top pod POD_NAME --containers
```

- Show metrics for a given pod and sort it by 'cpu' or 'memory:'

```
kubectl top pod POD_NAME --sort-by=cpu
```

Copying Files and Directories

These commands help you copy files and directories:

- Copy the /tmp/foo_dir local directory to /tmp/bar_dir in a remote pod in the current namespace:

```
kubectl cp /tmp/foo_dir my-pod:/tmp/bar_dir
```

- Copy the /tmp/foo local file to /tmp/bar in a remote pod in a specific container:

```
kubectl cp /tmp/foo my-pod:/tmp/bar -c my-container
```

- Copy the /tmp/foo local file to /tmp/bar in a remote pod in namespace my-namespace:

```
kubectl cp /tmp/foo my-namespace/my-pod:/tmp/bar
```

- Copy the /tmp/foo from a remote pod to /tmp/bar locally:

```
kubectl cp my-namespace/my-pod:/tmp/foo /tmp/bar
```

Interacting with Deployments and Services

This next set of commands help you interact (dump, listen, view, run) with deployments and services.

- Dump pod logs for a deployment (**single**-container case):

```
kubectl logs deploy/my-deployment
```

- Dump pod logs for a deployment (**multi**-container case):

```
kubectl logs deploy/my-deployment -c my-container
```

- Listen on local port 5000 and forward to port 5000 on Service backend

```
kubectl port-forward svc/my-service 5000
```

- Listen on local port 5000 and forward to Service target port with name <my-service-port>:

```
kubectl port-forward svc/my-service 5000:my-service-port
```

- Listen on local port 5000 and forward to port 6000 on a Pod created by <my-deployment>:

```
kubectl port-forward deploy/my-deployment 5000:6000
```

- Run command in first pod and first container in deployment (single- or multi-container cases):

```
kubectl exec deploy/my-deployment -- ls
```

Interacting with Nodes and Cluster

Learn how to interact with nodes and clusters with this next set of commands.

- Mark my-node as unschedulable:

```
kubectl cordon my-node
```

- Drain my-node to prepare for maintenance:

```
kubectl drain my-node
```

- Mark my-node as schedulable:

```
kubectl uncordon my-node
```

- Show metrics for a given node:

```
kubectl top node my-node
```

- Display addresses of the master and services:

```
kubectl cluster-info
```

- Dump current cluster state to stdout:

```
kubectl cluster-info dump
```

- Dump current cluster state to /path/to/cluster-state:

```
kubectl cluster-info dump --output-directory=/path/to/cluster-state
```

- Replace value as specified (if a taint with that key and effect already exists):

```
kubectl taint nodes foo dedicated=special-user:NoSchedule
```

Cluster Management

These commands help you manage clusters, from listing API resources to displaying pertinent information:

- Display endpoint information about the master and services in the cluster:

```
kubectl cluster-info
```

- Display the Kubernetes version running on the client and server:

```
kubectl version
```

- Retrieve the cluster configuration

```
kubectl config view
```

- List available API resources:

```
kubectl api-resources
```

- For listing the API versions that are available

```
kubectl api-versions
```

- List everything:

```
kubectl get all --all-namespaces
```

DaemonSet

A DaemonSet makes sure that nodes run pod copies. Nodes and pods are added to clusters. Similarly, pods undergo garbage collection once nodes are removed. When you delete a DaemonSet, all the pods created by it also get deleted.

Some typical DaemonSet uses for every node:

- running a cluster storage
- running a logs collection
- running a node monitoring

For simple cases, one DaemonSet could cover all nodes and each daemon type. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags, memory, and CPU requests for various hardware types.

```
Shortcode = ds
```

- List one or more DaemonSets:

```
kubectl get daemonset
```

- Edit and update the definition of one or more DaemonSet:

```
kubectl edit daemonset <daemonset_name>
```

- Delete a DaemonSet:

```
kubectl delete daemonset <daemonset_name>
```

- Create a new DaemonSet:

```
kubectl create daemonset <daemonset_name>
```

- Manage the rollout of a DaemonSet:

```
kubectl rollout daemonset
```

- Show the detailed state of DaemonSets within a namespace:

```
kubectl describe ds <daemonset_name> -n <namespace_name>
```

Deployments

A deployment runs multiple application replicas and automatically replaces any failed or unresponsive instances. Deployments are managed by the Kubernetes Deployment Controller. Moreover, deployments make sure that user requests are served through one or more instances of your application.

```
Shortcode = deploy
```

- List one or more deployments:

```
kubectl get deployment
```

- Show the detailed state of one or more deployments:

```
kubectl describe deployment <deployment_name>
```

- Edit and update the definition of one or more server deployment:

```
kubectl edit deployment <deployment_name>
```

- Create a new deployment:

```
kubectl create deployment <deployment_name>
```

- Delete the deployments:

```
kubectl delete deployment <deployment_name>
```

- Check the rollout status of a deployment:

```
kubectl rollout status deployment <deployment_name>
```

Events

A Kubernetes event is an object in the framework automatically generated in response to changes with other resources—like nodes, pods, or containers.

State changes lie at the center of this. For example, phases across a pod's lifecycle—like a transition from pending to running, or statuses like successful or failed—may trigger a Kubernetes event. The same goes for reallocations and scheduling.

```
Shortcode = ev
```

- List all recent events for all system resources:

```
kubectl get events
```

- List all warnings only:

```
kubectl get events --field-selector type=Warning
```

- List all events (excluding Pod events):

```
kubectl get events --field-selector involvedObject.kind!=Pod
```

- Pull all events for a single node with a specific name:

```
kubectl get events --field-selector involvedObject.kind=Node,
involvedObject.name=<node_name>
```

- Filter out normal events from a list of events:

```
kubectl get events --field-selector type!=Normal
```

Logs

System component logs record events happening in a cluster, which can be useful for debugging. To see the desired details of those events, you can configure log verbosity.

There can be two types of logs, namely fine-grained and coarse-grained. Coarse-grained logs represent errors within a component. On the other hand, fine-grained logs represent step-by-step traces of events.

- Print the logs for a pod:

```
kubectl logs <pod_name>
```

- Print the logs for the last hour for a pod:

```
kubectl logs --since=1h <pod_name>
```

- Retrieve the most recent 20 lines of logs:

```
kubectl logs --tail=20 <pod_name>
```

- Retrieve the logs from a service and optionally selecting which container:

```
kubectl logs -f <service_name> [-c <$container>]
```

- Print the logs for a pod and follow new logs:

```
kubectl logs -f <pod_name>
```

- Print the logs for a container in a pod:

```
kubectl logs -c <container_name> <pod_name>
```

- Get the output of the logs for a pod into a file named ‘pod.log:’

```
kubectl logs <pod_name> pod.log
```

- Check the logs for a previously failed pod:

```
kubectl logs --previous <pod_name>
```

Namespaces

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. Resource names must be unique *within* a namespace but not *across* namespaces. Namespace-based scoping is applicable only for namespaced objects (e.g., Deployments, Services, etc.) and not for cluster-wide objects (e.g., StorageClass, Nodes, PersistentVolumes, etc.).

```
Shortcode = ns
```

- Create a namespace <name>:

```
kubectl create namespace <namespace_name>
```

- List one or more namespaces:

```
kubectl get namespace <namespace_name>
```

- Display the detailed state of one or more namespace:

```
kubectl describe namespace <namespace_name>
```

- Delete a namespace:

```
kubectl delete namespace <namespace_name>
```

- Edit and update a namespace definition:

```
kubectl edit namespace <namespace_name>
```

- Display the Resource (CPU/Memory/Storage) usage for a namespace:

```
kubectl top namespace <namespace_name>
```

Replication Controllers

A replication controller is a key Kubernetes feature, responsible for:

- managing the pod lifecycle
- ensuring the specified number of pod replicas are running at any point of time
- Increases or decreasing the specified number of pods

```
Shortcode = rc
```

- List all the replication controllers:

```
kubectl get rc
```

- List the replication controllers by namespace:

```
kubectl get rc --namespace="<namespace_name>"
```

ReplicaSets

RepliceSets ensure you have a stable set of replica pods operating. You might use one to confirm that identical pods are available.

```
Shortcode = rs
```

- List all the ReplicaSets:

```
kubectl get replicasets
```

- Show the detailed state of one or more ReplicaSets:

```
kubectl describe replicasets <replicaset_name>
```

- Scale a ReplicaSet:

```
kubectl scale --replicas=[x]
```

Secrets

A secret is an object containing a small amount of sensitive data such as a password, token, or key. This information is either stored in a pod specification or in a container image. If you use a Secret, you don't need to include confidential data in your application code.

- Create a new secret:

```
kubectl create secret
```

- List all secrets:

```
kubectl get secrets
```

- List all the required details about secrets:

```
kubectl describe secrets
```

- Delete a secret:

```
kubectl delete secret <secret_name>
```

Services

Services are an abstract way to expose an application running on a set of Pods as a network service.

If you want to use any unfamiliar service discovery mechanism, there is no need to make changes to your application when developed using Kubernetes. This is because Kubernetes assigns each pod with a unique IP address and a single DNS can manage all the load across multiple pods.

```
Shortcode = svc
```

- List one or more services:

```
kubectl get services
```

- Show the detailed state of a service:

```
kubectl describe services
```

- Expose a replication controller, service, deployment, or pod as a new Kubernetes service:

```
kubectl expose deployment [deployment_name]
```

- Edit and update the definition of one or more services:

```
kubectl edit services
```

Service Accounts

A service account provides an identity for processes that run in a pod. Here is a useful introduction to Service Accounts.

```
Shortcode = sa
```

- List service accounts:

```
kubectl get serviceaccounts
```

- Show the detailed state of one or more service accounts:

```
kubectl describe serviceaccounts
```

- Replace a service account:

```
kubectl replace serviceaccount
```

- Delete a service account:

```
kubectl delete serviceaccount <service_account_name>
```

StatefulSet

StatefulSets represent a set of pods with unique, persistent identities and stable host names that GKE maintains regardless of where they are scheduled. The persistent disk storage associated with the StatefulSet is responsible for storing state information and other resilient data for the given StatefulSet pod.

```
Shortcode = sts
```

- List StatefulSet:

```
kubectl get statefulset
```

- Delete StatefulSet only (not pods):

```
kubectl delete statefulset/[stateful_set_name] --cascade=false
```