**Configmap and Secrets**

Every application needs different configuration. Anything that is configurable, changeable or varies between contexts should be submitted separately for each deployment that information needs to be supplied when the containers start . Context information can typically include names of other services, database locations, service URLs, running modes, feature enable/disable requests. Sensitive context information can include passwords, account IDs, security tokens etc.

This is where Kubernetes ConfigMaps and Secrets can help by supplying your deployment containers with the contextual and secretive information they require.

All the information related to configurations or secrets are stored in Etcd.

Pods provide containerized applications access to ConfigMaps and Secrets with three techniques:

1. command-line arguments
2. Environment variables
3. Files in a volume

**Step -1 :**

Created cluster ,nodegrp, then dashboard is deployed.

kubectl version --short && \ kubectl get componentstatus && \ kubectl get nodes && \ kubectl cluster-info

helm version –short

token.sh

**Step-2 :**

Before the containers can use the data it must first be stored in configmaps or secrets.

A ConfigMap is simple data associated with a unique key.

kubectl create configmap mountains --from-literal=aKey=aValue --from-literal=14ers=www.14ers.com

this will create a configmap with 2 values.

kubectl get configmap mountains

this will indicate 2 as we gave 2 values.

kubectl get configmap mountains -o yaml

apiVersion: v1

data:

14ers: www.14ers.com

aKey: aValue

kind: ConfigMap

metadata:

creationTimestamp: "2020-09-22T12:38:20Z"

managedFields:

- apiVersion: v1

fieldsType: FieldsV1

fieldsV1:

f:data:

.: {}

f:14ers: {}

f:aKey: {}

manager: kubectl

operation: Update

time: "2020-09-22T12:38:20Z"

name: mountains

namespace: default

resourceVersion: "912"

selfLink: /api/v1/namespaces/default/configmaps/mountains

uid: 5fdc939f-159d-4d5a-a5ad-385892c41e3f

* you can also use kubectl describe configmap mountains
* you can delete your configmap using kubectl delete configmap mountains

**Create config map from yaml :**

**ucs-org.yaml**

apiVersion: v1

kind: ConfigMap

metadata:

name: ucs-info

namespace: default

data:

property.1: hello

property.2: world

ucs-org: |-

description="Our scientists and engineers develop and implement innovative, practical solutions to some of our planet's most pressing problems"

formation=1969

headquarters="Cambridge, Massachusetts, US"

membership="over 200,000"

director="Kathleen Rest"

president="Kenneth Kimmell"

founder="Kurt Gottfried"

concerns="Global warming and developing sustainable ways to feed, power, and transport ourselves, to fighting misinformation, advancing racial equity, and reducing the threat of nuclear war."

website="ucsusa.org"

apply the configmap : kubectl apply -f ucs-org.yaml

describe the config map using : kubectl describe configmap ucs-info

**Three Access Techniques**

Once the configuration data is stored in ConfigMaps, the containers can access the data. Pods grant their containers access to the ConfigMaps through these three techniques:

* through the application command-line arguments,
* through the system environment variables accessible by the application,
* through a specific read-only file accessible by the application.

#### Command Line Arguments

passing in the data through the command-line arguments when running the container

apiVersion: v1

kind: Pod

metadata:

name: consume-via-cli

spec:

containers:

- name: consuming-container

image: k8s.gcr.io/busybox command: [ "/bin/sh", "-c", "echo $(PROPERTY\_ONE\_KEY); echo $(UCS\_INFO); env" ]

env:

- name: PROPERTY\_ONE\_KEY

valueFrom:

configMapKeyRef:

name: ucs-info

key: property.1

- name: UCS\_INFO valueFrom:

configMapKeyRef:

name: ucs-info

key: ucs-org

restartPolicy: Never

Using the Dashboard, inspect the Pod log and information page to see mapped data

#### Environment Variables

This example shows how a Pod accesses configuration data from the ConfigMap by passing in the data as environmental parameters of the container.

apiVersion: v1

kind: Pod

metadata: name: consume-via-env

spec:

containers:

- name: consuming-container

image: k8s.gcr.io/busybox

command: [ "/bin/sh", "-c", "env" ]

envFrom:

- configMapRef:

name: ucs-info

restartPolicy: Never

After apply command when the pod is ready you can check your logs to see the env variables :

kubectl logs consume-via-env

ouput:

KUBERNETES\_PORT=tcp://10.96.0.1:443

KUBERNETES\_SERVICE\_PORT=443

ucs-org=description="Our scientists and engineers develop and implement innovative, practical solutions to some of our planet's most pressing problems"

formation=1969

headquarters="Cambridge, Massachusetts, US"

membership="over 200,000"

director="Kathleen Rest"

president="Kenneth Kimmell"

founder="Kurt Gottfried"

concerns="Global warming and developing sustainable ways to feed, power, and transport ourselves, to fighting misinformation, advancing racial equity, and reducing the threat of nuclear war."

website="ucsusa.org"

HOSTNAME=consume-via-env

SHLVL=1

HOME=/root

KUBERNETES\_PORT\_443\_TCP\_ADDR=10.96.0.1

property.1=hello

property.2=world

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

KUBERNETES\_PORT\_443\_TCP\_PORT=443

KUBERNETES\_PORT\_443\_TCP\_PROTO=tcp

KUBERNETES\_PORT\_443\_TCP=tcp://10.96.0.1:443

KUBERNETES\_SERVICE\_PORT\_HTTPS=443

PWD=/

KUBERNETES\_SERVICE\_HOST=10.96.0.1

#### Volume Mounts :

This example shows how a Pod accesses configuration data from the ConfigMap by reading from a file in a directory of the container. Upon startup, the application would reference these parameters by referencing the named files in the known directory

apiVersion: v1

kind: Pod

metadata: name: consume-via-vol

spec:

containers:

- name: consuming-container

image: k8s.gcr.io/busybox

command: [ "/bin/sh","-c","cat /etc/config/keys" ]

volumeMounts:

- name: config-volume

mountPath: /etc/config

volumes:

- name: config-volume

configMap:

name: ucs-info

items:

- key: ucs-org

path: keys

restartPolicy: Never

Using the [Dashboard](https://2886795279-30000-ollie02.environments.katacoda.com/), inspect the Pod log and information page to see mapped data

**SECRETS:**kubectl create secret generic db-password --from-literal=password=MyDbPassw0rd

kubectl get secret db-password

kubectl get secret db-password -o json | jq

{

"apiVersion": "v1",

"data": {

"password": "TXlEYlBhc3N3MHJk"

},

"kind": "Secret",

"metadata": {

"creationTimestamp": "2020-09-22T12:55:02Z",

"managedFields": [

{

"apiVersion": "v1",

"fieldsType": "FieldsV1",

"fieldsV1": {

"f:data": {

".": {},

"f:password": {}

},

"f:type": {}

},

"manager": "kubectl",

"operation": "Update",

"time": "2020-09-22T12:55:02Z"

}

],

"name": "db-password",

"namespace": "default",

"resourceVersion": "3101",

"selfLink": "/api/v1/namespaces/default/secrets/db-password",

"uid": "916e9265-c4d7-45de-b4bd-480527b3dc88"

},

"type": "Opaque"

}

You can decode your password using : kubectl get secrets db-password -o 'go-template={{index .data "password"}}' | base64 --decode

**CREATE YOU SECRET USING YAML FILE :**

apiVersion: v1

kind: Secret

metadata:

name: db-creds

type: Opaque

data:

username: dXNlcgo=

password: TXlEYlBhc3N3MHJkCg==

you can give your password in encoded form by using echo MyDbPassw0rd | base64

or else ,

kubectl create secret generic db-password --from-literal=password=MyDbPassw0rd --dry-run -o yaml > my-secret.yaml

#### Read Secret

kubectl get secrets

db-creds Opaque 2 2m40s

db-password Opaque 1 6m1s

kubectl get secret db-password -o json | jq

how this can be accessed in pod : **kaurd.yaml :** kubectl apply -f kuard.yaml

apiVersion: v1

kind: Pod

metadata:

name: kuard

labels:

app: kuard

spec:

containers:

- image: gcr.io/kuar-demo/kuard-amd64:1

name: kuard

ports:

- containerPort: 8080

name: http

protocol: TCP

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

name: db-creds

key: username

- name: SECRET\_PASSWORD

valueFrom:

secretKeyRef:

name: db-creds

key: password

in this page the password and user are present :

<https://2886795279-31001-ollie02.environments.katacoda.com/-/env>

# **Injecting Vault Secrets Into Kubernetes Pods via a Sidecar**

* You can either install vault k8s using latest **vault helm chart**
* A [Docker image](https://hub.docker.com/r/hashicorp/vault-k8s) is also available. The Docker image can be used to manually run [vault-k8s](https://github.com/hashicorp/vault-k8s) within your scheduled environment if you choose not to use the Helm Chart.
* You can opt each application into Vault secret injection through the use of specifically set annotations within the pod configuration. Then, when the [vault-k8s](https://github.com/hashicorp/vault-k8s) webhook detects these specific annotations, it rewrites the pod definition based on what was requested
* You can leverage a selected namespace, specifically set annotations, and Kubernetes Service Accounts tied to a Vault Policy, this gives you fine-grained control of where and what secrets are injected without compromising on security.
* Use the [kubectl patch](https://kubernetes.io/docs/tasks/run-application/update-api-object-kubectl-patch/) command to apply the annotations to an existing Pod object, these will be intercepted by the [vault-k8s](https://github.com/hashicorp/vault-k8s) webhook service, which will then inject the correct init and sidecar containers along with the requested secrets.
* [Init](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-initialization/) and [Sidecar](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/). Init container to fetch secrets before an application starts, and a Sidecar container that starts alongside your application for keeping secrets fresh

**Patch-basic-annotations.yaml**

# patch-basic-annotations.yaml

spec:

template:

metadata:

annotations:

vault.hashicorp.com/agent-inject: "true"

vault.hashicorp.com/agent-inject-secret-helloworld: "secrets/helloworld"

vault.hashicorp.com/role: "myapp"

* **agent-inject** enables the Vault Agent injector service
* **role** is the Vault role created that maps back to the K8s service account
* **agent-inject-secret-FIlEPATH** prefixes the path of the file, database-config.txt written to /vault/secrets. The values is the path to the secret defined in Vault.

the above annotations when applied instructs [vault-k8s](https://github.com/hashicorp/vault-k8s) to inject an init and sidecar container into the requested Pod, fetch the secret/helloworld secret from Vault, and populate /vault/secrets/helloworld with that data, if the “myapp” role has access.

## Secrets Injection Workflow

Install vault k8s using vault helm chart.

**Step – 1 :**

First, before we install Vault, make sure injector support is enabled in the Vault Helm Chart values.yaml file.

injector:

enabled: true

**Step – 2 :**

Next, let’s configure a demo namespace, set the current context to it, and install Vault using the Helm Chart.

kubectl create namespace demo

kubectl config set-context --current --namespace=demo

helm install --name=vault

--set='server.dev.enabled=true' \

./vault-helm

**Step – 3 :**

Next connect to vault and configure policy named “app”

kubectl exec -ti vault-0 /bin/sh

cat <<EOF > /home/vault/app-policy.hcl

path "secret\*" {

capabilities = ["read"]

}

EOF

vault policy write app /home/vault/app-policy.hcl

**Step – 4 :**

Next configure **vault kubernetes auth** method and attach our newly recreated policy to our applications service account**.**

vault auth enable kubernetes

vault write auth/kubernetes/config \

token\_reviewer\_jwt="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \

kubernetes\_host=https://${KUBERNETES\_PORT\_443\_TCP\_ADDR}:443 \

kubernetes\_ca\_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt

vault write auth/kubernetes/role/myapp \

bound\_service\_account\_names=app \

bound\_service\_account\_namespaces=demo \

policies=app \

ttl=1h

**Step – 5 :**

Now lets create a sample secret using KV secrets engine .

vault kv put secret/helloworld username=foobaruser password=foobarbazpass

**Step – 6 :**

Deploy a sample application also create a service account to access the vault .

# app.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: app

labels:

app: vault-agent-demo

spec:

selector:

matchLabels:

app: vault-agent-demo

replicas: 1

template:

metadata:

annotations:

labels:

app: vault-agent-demo

spec:

serviceAccountName: app

containers:

- name: app

image: jweissig/app:0.0.1

---

apiVersion: v1

kind: ServiceAccount

metadata:

name: app

labels:

app: vault-agent-demo

kubectl create -f app.yaml

kubectl exec -ti app-XXXXXXXXX -c app -- ls -l /vault/secrets

here you can see that there are no secrets mounted .

Here is a annotations patch we can apply to our running example application’s pod configuration that sets specific annotations for injecting our secret/helloworld Vault secret.

# patch-basic-annotations.yaml

spec:

template:

metadata:

annotations:

vault.hashicorp.com/agent-inject: "true"

vault.hashicorp.com/agent-inject-secret-helloworld: "secrets/helloworld"

vault.hashicorp.com/role: "myapp"

kubectl patch deployment app --patch "$(cat patch-basic-annotations.yaml)"

kubectl exec -ti app-XXXXXXXXX -c app -- cat /vault/secrets/helloworld

**output :**

data: map[password:foobarbazpass username:foobaruser]

metadata: map[created\_time:2019-12-16T01:01:58.869828167Z deletion\_time: destroyed:false version:1]

when we applied the patch, our vault-k8s webhook intercepted and changed the pod definition, to include an Init container to pre-populate our secret, and a Vault Agent Sidecar to keep that secret data in sync throughout our applications lifecycle

**Output formatting : use** [Vault Agent Templates](https://www.vaultproject.io/docs/agent/template/index.html),

# patch-template-annotations.yaml

spec:

template:

metadata:

annotations:

vault.hashicorp.com/agent-inject: "true"

vault.hashicorp.com/agent-inject-status: "update"

vault.hashicorp.com/agent-inject-secret-helloworld: "secret/helloworld"

vault.hashicorp.com/agent-inject-template-helloworld: |

{{- with secret "secret/helloworld" -}}

postgresql://{{ .Data.data.username }}:{{ .Data.data.password }}@postgres:5432/wizard

{{- end }}

vault.hashicorp.com/role: "myapp"

kubectl patch deployment app --patch "$(cat patch-template-annotations.yaml)"

kubectl exec -ti app-XXXXXXXXX -c app -- cat /vault/secrets/helloworld

**output :**

postgresql://foobaruser:foobarbazpass@postgres:5432/wizard

**Installations done on cluster eks-30**

curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 >  
get\_helm.sh

chmod 700 get\_helm.sh

 ./get\_helm.sh

 helm –version

 helm repo add hashicorp <https://helm.releases.hashicorp.com>

 helm install vault hashicorp/vault

helm health status

 helm get status

 helm search repo 🡪 all helm charts will be displayed

 helm repo list

**History of Commands:**

 curl <https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3> > get\_helm.sh  
  601  chmod 700 get\_helm.sh  
  602  clear  
  603  ./get\_helm.sh  
  604  helm --version  
  605  helm help  
  606  clear  
  607  helm version  
  608  clear  
  609  history  
  610  helm repo add hashicorp <https://helm.releases.hashicorp.com>  
  611  helm install vault hashicorp/vault  
  612  helm status vault  
  613  helm get vault  
  614  clear  
  615  ls  
  616  helm get status  
  617  helm health status  
  618  clear  
  619  history  
  620  health status vault  
  621  helm status vault  
  622  helm get vault  
  623  ls  
  624  vi get\_helm.sh  
  625  clear  
  626  history  
  627  ls  
  628  helm repo list  
  629  helm view hashicrop  
  630  clear  
  631  helm view hashicorp  
  632  helm repo list  
  633  helm search repo  
  634  history  
  635  helm search repo  
  636  clear  
  637  kubectl -n kube-system get vaults  
  638  clear  
  639  kubectl get pods  
  640  clear  
  641  kubectl create namespace demo  
  642  kubectl get secretss  
  643  kubectl get secrets  
  644  helm search repo  
  645  helm repo list

**Links for vault :**

<https://learn.hashicorp.com/tutorials/vault/getting-started-dynamic-secrets>

<https://learn.hashicorp.com/tutorials/vault/database-secrets>

<https://www.vaultproject.io/docs/secrets/aws>

<https://learn.hashicorp.com/tutorials/vault/getting-started-dynamic-ssecrets>

**vault on to kubernetes server using helm and helm chart** :

<https://www.cloudops.com/blog/how-to-deploy-a-development-vault-server-to-kubernetes-using-helm/>

seal/unseal and modes of installations ha, dev :

<https://www.vaultproject.io/docs/platform/k8s/helm/run>

**installed hashicorp vault commands :**

helm install vault hashicorp/vault

helm status vault

helm get vault

**to go inside vault :** kubectl exec -ti vault-0 /bin/sh

**MODES IN VAULT :**

**DEV Mode :**

The Helm chart may run a Vault server in development. This installs a single Vault server with a memory storage backend.

**Stand alone mode :**

The Helm chart defaults to run in standalone mode. This installs a single Vault server with a file storage backend.

**HA Mode :**

The Helm chart may be run in high availability (HA) mode. This installs three Vault servers with an existing Consul storage backend. It is suggested that Consul is installed via the [Consul Helm chart](https://github.com/hashicorp/consul-helm).

**History :**

helm install vault hashicorp/vault --set "server.dev.enabled=true"

kubectl exec -ti vault-0 /bin/sh

kubectl logs vault-0

kubectl get pods -l app.kubernetes.io/name=vault

  765  vault status  
  767  kubectl exec -ti vault-0 bin/sh  
  768  kubectl exec -ti vault-0 -- vault operator unseal  
  770  kubectl exec -ti vault-0 -- vault operator init

  772  vault secret list  
  773  kubectl exec -ti vault-0 bin/sh  
  774  kubectl get pods  
  775  helm uninstall vault

  777  helm install vault hashicorp/vault  
  
  779  kubectl get pods  
  780  kubectl get pods -l app.kubernetes.io/name=vault

  783  kubectl exec -ti vault-0 -- vault operator init  
  784  kubectl exec -ti vault-0 -- vault operator unseal (enter keys which we get from 783 command )

  785  kubectl exec -ti vault-0 -- vault operator unseal  
  786  kubectl get pods -l app.kubernetes.io/name=vault

789  vault login (give root token get that token using **kubectl logs vault-0**)  
  790  kubectl exec -ti vault-0 -- bin/sh

792  helm uninstall vault  
  794  helm install vault hashicorp/vault --set "server.dev.enabled=true"  
  795  helm get vault  
 801  kubectl logs vault-0  
  802  kubectl exec -ti vault-0 /bin/sh

**Enabling aws secrets engine :**

* vault secrets enable -path=aws aws
* vault write aws/config/root \

access\_key=ACCESS\_KEY\_ID \

secret\_key=SECRET\_ACCESS\_KEY

* vault write aws/roles/my-role \

credential\_type=iam\_user \

policy\_document=-<<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": "ec2:\*",

"Resource": "\*"

}

]

}

EOF

* vault read aws/creds/my-role

Key Value

--- -----

lease\_id aws/creds/my-role/e75QUrah9HuVm3c5KUBJaKg9

lease\_duration 768h

lease\_renewable true

access\_key AKIA4OJTSXOYK5G5ODEH

secret\_key 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

security\_token <nil>

/ $ vault read aws/creds/my-role

* $ vault read aws/creds/my-role

Key Value

--- -----

lease\_id aws/creds/my-role/BaKLF4ttzoQiQfbuTE9Lz2g3

lease\_duration 768h

lease\_renewable true

access\_key AKIA4OJTSXOYOG3OODNL

secret\_key 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

security\_token <nil>

* vault lease revoke aws/creds/my-role/eHKMYIA7HbgRUC0uLhIsqwdC 🡪this will delete the role created.
* vault write aws/config/lease lease=24h lease\_max=24h