Deploying Kubernetes clusters with GitOps and ArgoCD

VMware Modern Applications

Deploying Kubernetes clusters with GitOps and ArgoCD

Introduction

vSphere with Tanzu has the ability to deploy K8s clusters via K8s CustomResourceDefinitions (CRDs) - kind of like K8s, managing K8s - if you like. For those new to the space, this might seem a little arcane - but there are actually very good reasons why we built it this way. For starters, it lets you define your K8s clusters *declaratively*, just like how you define your K8s workloads, the benefit to this is that your K8s clusters will constantly be reconciled against the desired state stored in the K8s API, including through upgrades, expansions, shrink operations, etc.

One of the major advantages to managing K8s clusters through K8s CRDs, is the ability to use a GitOps model to manage your entire estate, including workload configuration, secrets, even the K8s clusters themselves! The high level pitch is that GitOps allows you to manage your entire app stack, or indeed anything managed through the K8s API in a revisioned, declarative, scalable manner that auto-heals.

Sounds cool? Good! Let's take a look at how to do it with ArgoCD and vSphere with Tanzu.

Requirements

We are going to assume you have a few things already set up:

- An ArgoCD instance (probably deployed on a TKG cluster)
- vSphere with Tanzu on vSphere 7.0 U3+
- K8s CLI tools (`kubectl`, `helm`)
- Some standard CLI tools (`jq`, `xargs`, `base64`)

To allow ArgoCD to manage K8s clusters on vSphere with Tanzu, it needs to be able to access the Supervisor cluster API and authenticate with it - in 7.0 U3 we added the ability to create K8s `ServiceAccounts` and `RoleBindings` on the Supervisor cluster, which allows ArgoCD to authenticate with the Supervisor cluster using a consistent non-changing token.

This `ServiceAccount` is created in the vSphere with Tanzu Namespace that you want ArgoCD to manage objects inside using your regular login and `kubectl` so let's take a look at how to do that first.

Authenticate with the Supervisor cluster

The first thing we need to do is authenticate ourselves with the Supervisor cluster and target the `Namespace` we're going to use for ArgoCD. So let's login:

```
$ kubectl vsphere login --server https://10.198.53.128 --insecure-skip-tls-verify -u administrator@vsphere.local

Logged in successfully.

You have access to the following contexts:
   10.198.53.128
   myles-ns-01-10.198.53.128

If the context you wish to use is not in this list, you may need to try
logging in again later, or contact your cluster administrator.
```

Let's change into the `Namespace` that we're targeting (pro tip: `kubectx/kubens` are great tools for managing contexts quickly):

```
$ kubectl config use-context myles-ns-01-10.198.53.128
Switched to context "myles-ns-01-10.198.53.128".
```

Create a ServiceAccount

Now that we're targeting the `Namespaces` we want to use, let's create a `ServiceAccount` and a `RoleBinding` for ArgoCD to use:

```
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  namespace: myles-ns-01
  name: argocd-robot
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: argocd-edit-binding
  namespace: myles-ns-01
subjects:
- kind: ServiceAccount
  name: argocd-robot
  namespace: myles-ns-01
  apiGroup: ""
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: ""
EOF

serviceaccount/argocd-robot created
rolebinding.rbac.authorization.k8s.io/argocd-edit-binding configured
```

We have just created an account for ArgoCD to use, and K8s will automatically create a secret token for the account and store it in a `Secret` object for us to retrieve and use later.

The second part of the manifest is the `RoleBinding`, this takes the `ServiceAccount` (`argocd-robot`) and binds it to an already existing `ClusterRole` on the cluster - in the spirit of least privilege, we're binding it to the `edit` role - but you could equally bind it to the `cluster-admin` role should you so wish.

Retrieve the Secret token

At this stage, we can retrieve the `Secret` that K8s created for the `ServiceAccount` and use it to build out an authentication `Secret` that ArgoCD will use to target this `Namespace` with:

```
$ kubectl get serviceaccounts argocd-robot -o json | jq -r '.secrets[] .name' | xargs -I {} sh -c "kubectl get secret
-o json {} | jq -r '.data .token'" | base64 -d

eyJhbGciOiJSUzI1NiIsImtpZCI6Imw3V3RXLS1jQ3YzTTJVbVMtVXl0a0loT2NTRmVYZTFqUjFtX2w0TnM4d1kifQeyJpc3MiOiJrdWJlcm5ldGVzL3Nl
cnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlY-rest-of-the-token-here-
rHKrcxCguqbx8d0O_QNX7eoyCuWjHEd4mg2r49lpFq_vaGjvnPVzYfIph15k7fgNCgX2Dxti0GtZ_fzVKoQ0rJY035f9awE0X7vkihSI6zoOclzFfev-
_ldvLbmmxy2tYPsEbRNU4WTL4REdNBkN7A
```

That looks like a lot - but bit by bit its quite simple - we retrieve the `ServiceAccount` details and output that to JSON, query the JSON with `jq` and get the name of the `Secret` associated with the account - then we use `xargs` to run a shell command to retrieve the `Secret` data and output it to JSON, then we use `jq` again to the get the token and decode the base64 encoded `Secret` data and output it to the shell.

This is the token that we need to use in our ArgoCD config.

Constructing the ArgoCD config

ArgoCD allows you to feed it configuration for clusters via a `Secret` object so let's create that on the cluster and namespace where ArgoCD is running - for me, i've deployed ArgoCD onto a TKG cluster, so i'll change to that context:

```
kubectl config use-context tkc-3
```

Change into the namespace that you have ArgoCD deployed into:

```
kubens argocd
```

First off, we need to build the configuration that ArgoCD is expecting, we need the token from the `ServiceAccount` we created earlier as well as the `Namespace` that we want to target and the `Server` URL that we're going to use to connect to the cluster - in my case that info looks like this:

- Token: `eyJhbGci........BkN7A`
- Namespace: `myles-ns-01`
- Server: `https://10.198.53.128:6443`

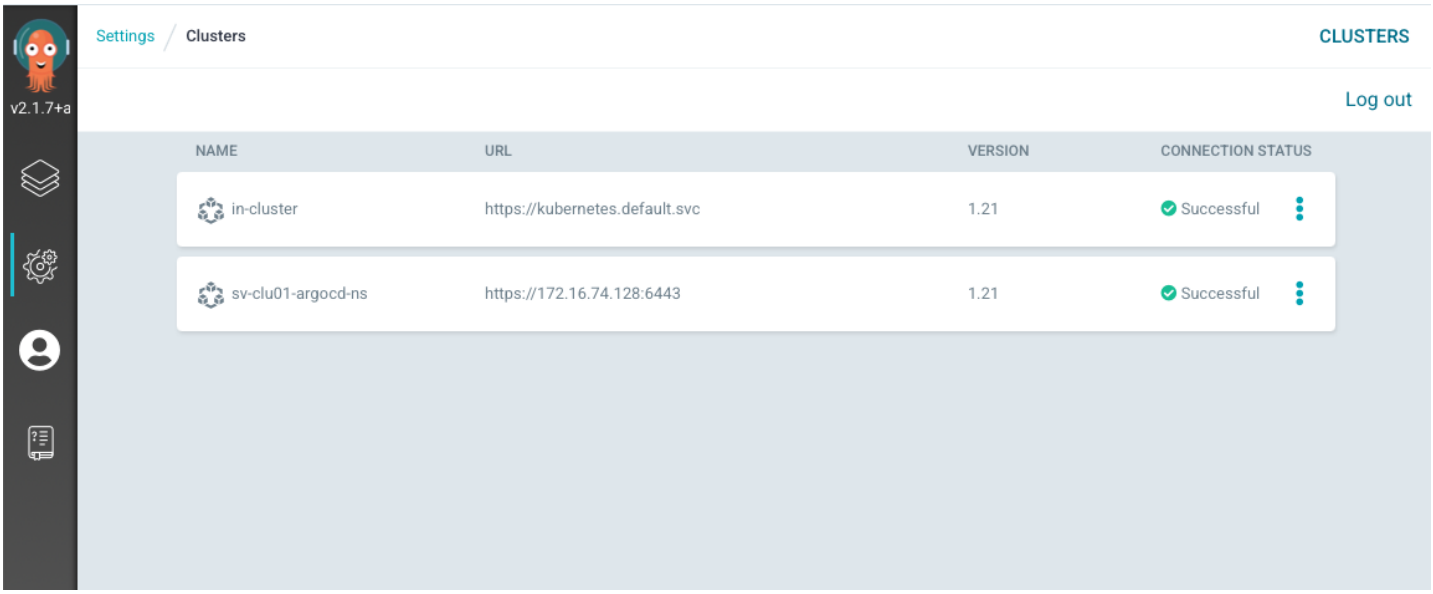
We need to plug this info into the below format for ArgoCD (your token will be a lot longer, I've shortened mine for brevity):

```
{
  "bearerToken": "eyJhbGciOiJSUzI1NiIsI......EdNBkN7A",
  "tlsClientConfig": {
    "insecure": true
  }
}
```

Let's build out the `Secret` that ArgoCD will use to connect to the cluster - of particilar note here is the label `argocd.argoproj.io/secret-type: cluster` this is what tells ArgoCD that it should use this object to connect to the cluster (the `stringData.name` can be whatever you want the cluster to be referred to within ArgoCD):

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  labels:
    argocd.argoproj.io/secret-type: cluster
  name: sv-cluster-argocd-ns
  namespace: argocd
type: Opaque
stringData:
  config: |
    {
    "bearerToken": "eyJhbGciOiJSUz......_fg6jiqwUQZLUrg",
        "tlsClientConfig": {
        "insecure": true
        }
    }
  name: sv-clu01-argocd-ns
  namespaces: myles-ns-01
  server: https://10.198.53.128:6443
EOF
```

With the `Secret` created - you should now see the Namespace show up in your ArgoCD instance - not to worry if it doesn't show `Successful` yet, we have to make some changes to the ArgoCD config to make it honour the Supervisor cluster's restrictive RBAC policies.



Configuring ArgoCD

Because the Supervisor cluster uses a least-privilege approach to permissions, we need to configure ArgoCD to only reconcile and look for the `Resources` and `CustomResourceDefinitions` that we want to allow access to - i've pre-created a list of CRDs that would be reasonable to allow access to and will let you configure pretty much everything through GitOps.

In the `argocd-cm ConfigMap` object that holds the configuration for the ArgoCD server, add the following keys to the `data` section (replacing `https://10.198.53.128:6443` with your SV cluster server URL from above) - if you are using `Helm` you can find my config here:

```
resource.exclusions: |
    - apiGroups:
    - "*"
    kinds:
    - PodTemplate
    clusters:
    - https://10.198.53.128:6443
resource.inclusions: |
    - apiGroups:
    - "run.tanzu.vmware.com"
    kinds:
    - TanzuKubernetesAddon
    - TanzuKubernetesCluster
    - TanzuKubernetesRelease
    - TkgServiceConfiguration
    clusters:
    - https://10.198.53.128:6443
    - apiGroups:
    - "cluster.x-k8s.io"
    kinds:
    - Cluster
    - Machine
    - MachineDeployment
    - MachineHealthCheck
    - MachineSet
    clusters:
    - https://10.198.53.128:6443
    - apiGroups:
    - "vmoperator.vmware.com"
    kinds:
    - "*"
    clusters:
    - https://10.198.53.128:6443
    - apiGroups:
    - "controlplane.cluster.x-k8s.io"
    - "bootstrap.cluster.x-k8s.io"
```

```
    kinds:
    - KubeadmControlPlane
    - KubeadmConfig
    - KubeadmConfigTemplate
    clusters:
    - https://10.198.53.128:6443
    - apiGroups:
    - "infrastructure.cluster.vmware.com"
    kinds:
    - "*"
    clusters:
    - https://10.198.53.128:6443
    - apiGroups:
    - "rbac.authorization.k8s.io"
    kinds:
    - Role
    - RoleBinding
    clusters:
    - https://10.198.53.128:6443
```

You may need to delete the ArgoCD server pod to get it to take the new config:

```
kubectl delete pod -l app.kubernetes.io/name=argocd-server
```

Creating TKG Clusters with ArgoCD

At this stage ArgoCD should now be successfully connected to the Supervisor cluster Namespace and we are now able to actually deploy a TKG cluster with it!

ArgoCD requires two things to manage K8s apps, the `Application` object and the manifests you wish to deploy - I have a full GitOps repo that runs on my Tanzu cluster that you can see the config for here but to keep things simple, we're going to start off with a single `Application` and it will just point to a folder from which we will deploy all manifests.

The `Application` object tells ArgoCD where to look for manifests, and what cluster to deploy and manage them on - so let's create one that targets our SV Namespace - the comments i've added will explain what the critical sections do and you should change them to suit your needs:

```
kubectl apply -f - <<EOF
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: tanzu-clusters
  namespace: argocd
  annotations:
    argocd.argoproj.io/sync-wave: "0"
  finalizers:
    - resources-finalizer.argocd.argoproj.io
spec:
  project: default
  source:
    # The repository that your TKC manfiests are stored in, i'm using my GitOps repo here
    repoURL: https://github.com/mylesagray/tanzu-cluster-gitops.git
    # What version do you want ArgoCD to deploy from the repo? HEAD is the latest
    targetRevision: HEAD
    # What folder in the repo do you want to deploy manifests from?
    # You can see mine here: https://github.com/mylesagray/tanzu-cluster-gitops/tree/master/manifests/tanzu-clusters
    path: manifests/tanzu-clusters
    # What type of folder is this, and do we want to recurse into subfolders?
    # ArgoCD supports many types, including Helm, jsonnet, bare YAML, and more.
    # In this example we're using bare YAML to keep it simple, but my GitOps repo above has
    # many more complex examples and uses ArgoCD's app-of-apps pattern if you want them.
    directory:
      recurse: true
  destination:
    # The cluster to deploy the manifests to - this is the Supervisor cluster
    server: 'https://10.198.53.128:6443'
    # The Namespace to deploy the manifests to - this is the Supervisor cluster Namespace
    namespace: myles-ns-01
  syncPolicy:
```

```
    # Allow ArgoCD to automatically clean up leftovers if the manifests are removed
    # and self-heal any issues that arise
    automated:
      prune: true
      selfHeal: true
EOF
```

As you can see from the above, we are telling ArgoCD to deploy from the `manifests/tanzu-clusters` folder in my repo, and to deploy to the `myles-ns-01` Namespace on the Supervisor cluster.

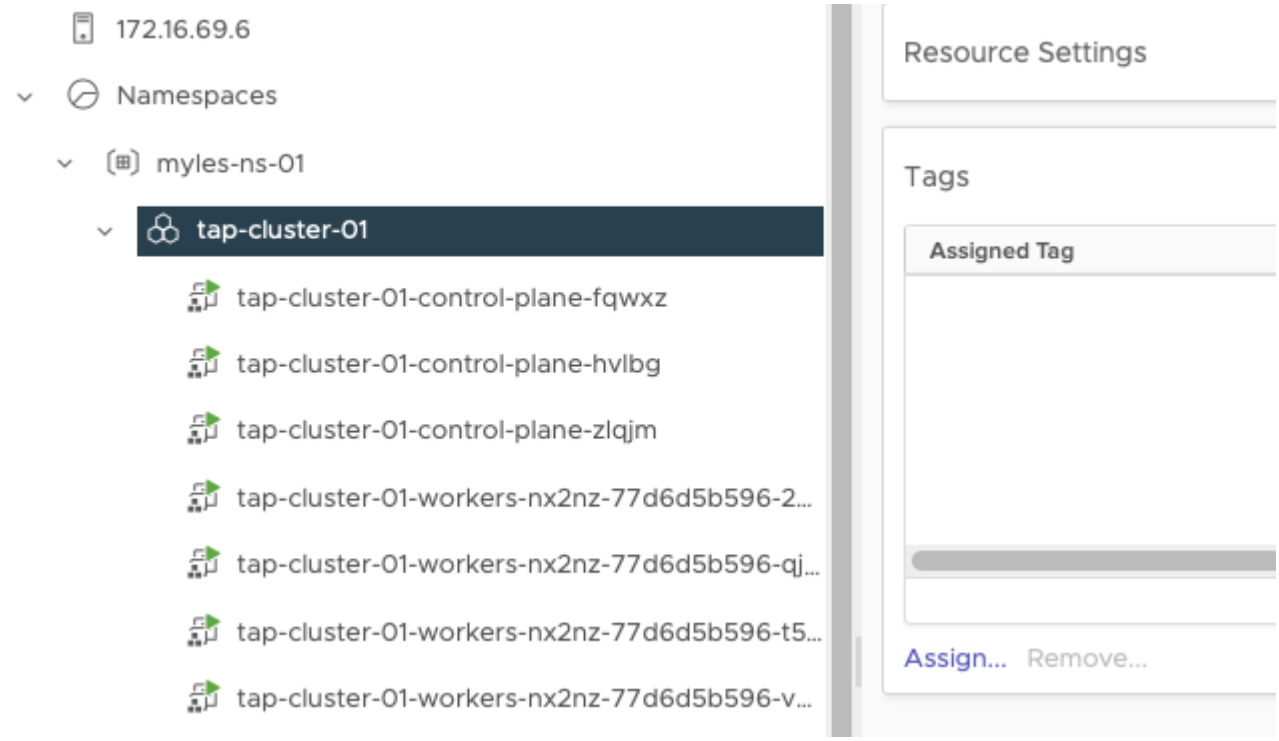
Inside the `manifests/tanzu-clusters` folder is a single manifest that defines a TKC:

```
apiVersion: run.tanzu.vmware.com/v1alpha2
kind: TanzuKubernetesCluster
metadata:
  name: tap-cluster-01
  namespace: myles-ns-01
spec:
  distribution:
    version: v1.21
  settings:
    storage:
      classes:
      - vsan-default-storage-policy
      defaultClass: vsan-default-storage-policy
  topology:
    controlPlane:
      replicas: 3
      storageClass: vsan-default-storage-policy
      vmClass: best-effort-large
    nodePools:
    - name: workers
      replicas: 4
      storageClass: vsan-default-storage-policy
      vmClass: best-effort-large
```

You obviously can, and should - change the above to meet your environment specs. But if you choose to deploy my manifest using my repo as the source, that's fine - but be aware, any time I change my GitOps repo, ArgoCD will sync the changes from the repo to your cluster - so as an example, if I delete the `tap-cluster-01.yaml` manifest from that folder, the TKC will be deleted from your environment.

It is highly advisable that you fork my repo and instead, target the fork in your ArgoCD config so that you have full control over everything and can see how adding and removing, or adjusting things results in ArgoCD updating your environment to match that state.

If we've done everything correctly - you should now have a TKC deployed in your SV cluster, matching the spec you had in your manifest:



As you update the manifest in your repo - ArgoCD will automatically reconcile the changes to the cluster, so try adding, removing or changing node sizes to get a feel for the workflow!

If you have any questions - feel free to reach out to @mylesagray on Twitter.