Manage Kubernetes Resources via Terraform

22min

Add bookmark

Kubernetes (K8S) is an open-source workload scheduler with focus on containerized applications. You can use the Terraform Kubernetes provider to interact with resources supported by Kubernetes.

In this tutorial, you will learn how to interact with Kubernetes using Terraform, by scheduling and exposing a NGINX deployment on a Kubernetes cluster. You will also manage custom resources using Terraform.

The final Terraform configuration files used in this tutorial can be found in the Deploy NGINX on Kubernetes via Terraform GitHub repository.

Why deploy with Terraform?

While you could use `kubectl` or similar CLI-based tools to manage your Kubernetes resources, using Terraform has the following benefits:

- **Unified Workflow** - If you are already provisioning Kubernetes clusters with Terraform, use the same configuration language to deploy your applications into your cluster.
- **Full Lifecycle Management** - Terraform doesn't only create resources, it updates, and deletes tracked resources without requiring you to inspect the API to identify those resources.
- **Graph of Relationships** - Terraform understands dependency relationships between resources. For example, if a Persistent Volume Claim claims space from a particular

Persistent Volume, Terraform won't attempt to create the claim if it fails to create the volume.

Prerequisites

The tutorial assumes some basic familiarity with Kubernetes and `kubectl`.

It also assumes that you are familiar with the usual Terraform plan/apply workflow. If you're new to Terraform itself, refer first to the Getting Started tutorial.

For this tutorial, you will need an existing Kubernetes cluster. If you don't have a Kubernetes cluster, you can use kind to provision a local Kubernetes cluster or provision one on a cloud provider.

kind | AWS (EKS) | Azure (AKS) | Google Cloud (GKE)

Follow these instructions or choose a package manager based on your operating system to install kind.

macOS install with Homebrew | Windows install with Chocolatey

Use the package manager `homebrew` to install kind.

```
$ brew install kind
```

Once you've done this, download and save the kind configuration into a file named `kind-config.yaml`. This configuration adds extra port mappings, so you can access the NGINX service locally later.

```
$ curl https://raw.githubusercontent.com/hashicorp/learn-terraform-deploy-ngin
```

Then, create a kind Kubernetes cluster.

```
$ kind create cluster --name terraform-learn --config kind-config.yaml
Creating cluster "terraform-learn" ...
 ✓ Ensuring node image (kindest/node:v1.17.0) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-terraform-learn"
You can now use your cluster with:

kubectl cluster-info --context kind-terraform-learn

Have a nice day! 👋
```

Verify that your cluster exists by listing your kind clusters.

```
$ kind get clusters
terraform-learn
```

Then, point `kubectl` to interact with this cluster. The context is `kind-` followed by the name of your cluster.

```
$ kubectl cluster-info --context kind-terraform-learn
Kubernetes master is running at https://127.0.0.1:32769
KubeDNS is running at https://127.0.0.1:32769/api/v1/namespaces/kube-system/se

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump
```

Configure the provider

Before you can schedule any Kubernetes services using Terraform, you need to configure the Terraform Kubernetes provider.

There are many ways to configure the Kubernetes provider. We recommend them in the following order (most recommended first, least recommended last):

1. Use cloud-specific auth plugins (for example, `eks get-token`, `az get-token`, `gcloud config`)
2. Use oauth2 token
3. Use TLS certificate credentials
4. Use `kubeconfig` file by setting **both** `config_path` and `config_context`
5. Use username and password (HTTP Basic Authorization)

Follow the instructions in the kind or cloud provider tabs to configure the provider to target a specific Kubernetes cluster. The cloud provider tabs will configure the Kubernetes provider using cloud-specific auth tokens.

Create a directory named `learn-terraform-deploy-nginx-kubernetes`.

```
$ mkdir learn-terraform-deploy-nginx-kubernetes
```

Then, navigate into it

```
$ cd learn-terraform-deploy-nginx-kubernetes
```

Note

This directory is **only** used for managing Kubernetes cluster resources with Terraform. By keeping the Terraform configuration for provisioning a Kubernetes cluster and managing a Kubernetes resources separate, changes in one repository doesn't affect the other. In addition, the modularity makes the configuration more readable and enables you to scope different permissions to each workspace.

kind | AWS (EKS) | Azure (AKS) | Google Cloud (GKE)

Then, create a new file named `kubernetes.tf` and add the following configuration to it. This serves as a base configuration for the provider.

```
terraform {
  required_providers {
    kubernetes = {
      source = "hashicorp/kubernetes"
    }
  }
}

variable "host" {
  type = string
}

variable "client_certificate" {
  type = string
}

variable "client_key" {
  type = string
}

variable "cluster_ca_certificate" {
  type = string
}

provider "kubernetes" {
  host = var.host

  client_certificate     = base64decode(var.client_certificate)
  client_key             = base64decode(var.client_key)
  cluster_ca_certificate = base64decode(var.cluster_ca_certificate)
}
```

To properly configure this provider, you need to define the variables.

First, view your kind cluster information.

```
$ kubectl config view --minify --flatten --context=kind-terraform-learn
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRU...
    server: https://127.0.0.1:32768
  name: kind-terraform-learn
contexts:
- context:
    cluster: kind-terraform-learn
    user: kind-terraform-learn
  name: kind-terraform-learn
current-context: kind-terraform-learn
kind: Config
preferences: {}
users:
- name: kind-terraform-learn
  user:
    client-certificate-data: LS0tLS1CRU...
    client-key-data: LS0tLS1CRU...
```

Define the variables in a `terraform.tfvars` file.

- `host` corresponds with `clusters.cluster.server`.
- `client_certificate` corresponds with `users.user.client-certificate`.
- `client_key` corresponds with `users.user.client-key`.
- `cluster_ca_certificate` corresponds with `clusters.cluster.certificate-authority-data`.

You should end up with something similar to the following.

terraform.tfvars

```
host                   = "https://127.0.0.1:32768"
client_certificate     = "LS0tLS1CRUdJTiB..."
client_key             = "LS0tLS1CRUdJTiB..."
cluster_ca_certificate = "LS0tLS1CRUdJTiB..."
```

After configuring the provider, run `terraform init` to download the latest version and initialize your Terraform workspace.

```
$ terraform init
```

Schedule a deployment

Add the following to your `kubernetes.tf` file. This Terraform configuration will schedule a NGINX deployment with two replicas on your Kubernetes cluster, internally exposing port 80 (HTTP).

kubernetes.tf

```
resource "kubernetes_deployment" "nginx" {
  metadata {
    name = "scalable-nginx-example"
    labels = {
      App = "ScalableNginxExample"
    }
  }

  spec {
    replicas = 2
    selector {
      match_labels = {
        App = "ScalableNginxExample"
      }
    }
    template {
      metadata {
        labels = {
          App = "ScalableNginxExample"
        }
      }
      spec {
        container {
          image = "nginx:1.7.8"
          name  = "example"

          port {
            container_port = 80
          }

          resources {
            limits = {
              cpu    = "0.5"
              memory = "512Mi"
            }
            requests = {
              cpu    = "250m"
              memory = "50Mi"
```

```
      }
    }
  }
}
```

You may notice the similarities between the Terraform configuration and Kubernetes configuration YAML file.

Apply the configuration to schedule the NGINX deployment. Confirm your apply with a `yes`.

```
$ terraform apply

Terraform used the selected providers to generate the following execution plan. Re
indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # kubernetes_deployment.nginx will be created
  + resource "kubernetes_deployment" "nginx" {
      ## ...
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

kubernetes_deployment.nginx: Creating...
kubernetes_deployment.nginx: Still creating... [10s elapsed]
kubernetes_deployment.nginx: Still creating... [20s elapsed]
kubernetes_deployment.nginx: Creation complete after 26s [id=default/scalable-ngin

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

```
$ kubectl get deployments
NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
scalable-nginx-example   2/2     2            2           15s
```

Schedule a Service

There are multiple Kubernetes services you can use to expose your NGINX to users.

If your Kubernetes cluster is hosted locally on kind, you will expose your NGINX instance via **NodePort** to access your instance. This exposes the service on each node's IP at a static port, allowing you to access the service from outside the cluster at `<NodeIP>:<NodePort>`.

If your Kubernetes cluster is hosted on a cloud provider, you will expose your NGINX instance via **LoadBalancer** to access your instance. This exposes the service externally using a cloud provider's load balancer.

Notice how the Kubernetes Service resource block dynamically assigns the selector to the Deployment's label. This avoids common bugs due to mismatched service label selectors.

kind | AWS (EKS) | Azure (AKS) | Google Cloud (GKE)

Add the following configuration to your `kubernetes.tf` file. This will expose the NGINX instance at the `node_port`: `30201`.

kubernetes.tf

```
resource "kubernetes_service" "nginx" {
  metadata {
    name = "nginx-example"
  }
  spec {
    selector = {
      App = kubernetes_deployment.nginx.spec.0.template.0.metadata[0].labels.A
    }
    port {
      node_port   = 30201
      port        = 80
      target_port = 80
    }

    type = "NodePort"
  }
}
```

Apply the configuration to schedule the NodePort Service. Confirm your apply with a `yes`.

```
$ terraform apply
kubernetes_deployment.nginx: Refreshing state... [id=default/scalable-nginx-ex

## ...

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

kubernetes_service.nginx: Creating...
kubernetes_service.nginx: Creation complete after 0s [id=default/nginx-example
```

Once the apply is complete, verify the NGINX service is running.

```
$ kubectl get services
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1      <none>        443/TCP        2m53s
nginx-example   NodePort    10.96.55.64    <none>        80:30201/TCP   76s
```

You can access the NGINX instance by navigating to the NodePort at `http://localhost:30201/`.

Scale the deployment

You can scale your deployment by increasing the `replicas` field in your configuration. Change the number of replicas in your Kubernetes deployment from `2` to `4`.

kubernetes.tf

```
resource "kubernetes_deployment" "nginx" {
  ## ...

  spec {
    replicas = 4

    ## ...
  }

  ## ...
}
```

Apply the change to scale your deployment. Confirm your apply with a `yes`.

```
$ terraform apply
kubernetes_deployment.nginx: Refreshing state... [id=default/scalable-nginx-exampl
kubernetes_service.nginx: Refreshing state... [id=default/nginx-example]

## ...

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

kubernetes_deployment.nginx: Modifying... [id=default/scalable-nginx-example]
kubernetes_deployment.nginx: Modifications complete after 0s [id=default/scalable-

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Once the apply is complete, verify the NGINX deployment has four replicas.

```
$ kubectl get deployments
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
scalable-nginx-example   4/4     4            4           4m48s
```

Managing Custom Resources

In addition to built-in resources and data sources, the Terraform provider also includes a `kubernetes_manifest` resource that lets you manage custom resource definitions (CRDs), custom resources, or any resource that is not built into the Terraform provider.

You will use Terraform to apply a CRD then manage custom resources. You have to do this in two steps:

1. Apply the required CRD to the cluster

You need two apply steps because at plan time Terraform queries the Kubernetes API to verify the schema for the kind of object specified in the `manifest` field. If Terraform doesn't find the CRD for the resource defined in the manifest the plan will return an error.

> **Note**
> To make this tutorial faster we included the CRD in the same workspace as the Kubernetes resources that it manages. In production create a new workspace for the CRD.

Create a custom resource definition

Create a new file named `crontab_crd.tf` and paste in the bellow configuration for a CRD that extends Kubernetes to store cron data as a resource called CronTab.

crontab_crd.tf

```
resource "kubernetes_manifest" "crontab_crd" {
  manifest = {
    "apiVersion" = "apiextensions.k8s.io/v1"
    "kind"       = "CustomResourceDefinition"
    "metadata" = {
      "name" = "crontabs.stable.example.com"
    }
    "spec" = {
      "group" = "stable.example.com"
      "names" = {
        "kind"   = "CronTab"
        "plural" = "crontabs"
        "shortNames" = [
          "ct",
        ]
        "singular" = "crontab"
      }
      "scope" = "Namespaced"
      "versions" = [
        {
          "name" = "v1"
          "schema" = {
            "openAPIV3Schema" = {
              "properties" = {
                "spec" = {
                  "properties" = {
                    "cronSpec" = {
                      "type" = "string"
                    }
                    "image" = {
                      "type" = "string"
                    }
                  }
                  "type" = "object"
                }
              }
              "type" = "object"
            }
```

```
        },
      ]
    }
  }
}
```

The resource has two configurable fields: `cronSpec` and `image`. Apply the configuration to create the CRD. Confirm your apply with `yes`.

```
$ terraform apply

Terraform used the selected providers to generate the following execution plan. Re
actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # kubernetes_manifest.crontab_crd will be created
  + resource "kubernetes_manifest" "crontab_crd" {
      + manifest = {
          # ...
        }
      + object   = {
          # ...
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

kubernetes_manifest.crontab_crd: Creating...
kubernetes_manifest.crontab_crd: Creation complete after 0s
```

Note that in the plan, Terraform created a resource with two attributes: `manifest` and `object`.

1. The `manifest` attribute is your desired configuration, and `object` is the end state returned by the Kubernetes API server after Terraform created the resource.
2. The `object` attribute contains many more fields than you specified in `manifest` because Terraform generated a schema containing all of the possible resource attributes that the Kubernetes API server could add. When referencing the `kubernetes_manifest` resource from outputs or other resources, always use the `object` attribute.

Confirm that Terraform created the CRD using `kubectl`.

```
$ kubectl get crds crontabs.stable.example.com
```


The `contrabs` resource definition now exists in Kubernetes, but you have not used it to define any Kubernetes resources yet. Check for the resource definition with `kubectl`, which would return `error: the server doesn't have a resource type "crontab"` if the CRD didn't exist.

```
$ kubectl get crontabs
No resources found in default namespace.
```

Create a custom resource

Now, create a new file named `my_new_crontab.tf` and paste in the following configuration, which creates a custom resource based on your newly created CronTab CRD.

my_new_crontab.tf

```
resource "kubernetes_manifest" "my_new_crontab" {
  manifest = {
    "apiVersion" = "stable.example.com/v1"
    "kind"       = "CronTab"
    "metadata" = {
      "name"      = "my-new-cron-object"
      "namespace" = "default"
    }
    "spec" = {
      "cronSpec" = "* * * * */5"
      "image"    = "my-awesome-cron-image"
    }
  }
}
```

Apply the configuration to create the custom resource. Confirm the apply with `yes`.

```
$ terraform apply

Terraform used the selected providers to generate the following execution plan. Re
actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # kubernetes_manifest.my_new_crontab will be created
  + resource "kubernetes_manifest" "my_new_crontab" {
      + manifest = {
          + apiVersion = "stable.example.com/v1"
          + kind       = "CronTab"
          + metadata   = {
              + name      = "my-new-cron-object"
              + namespace = "default"
            }
          + spec       = {
              + cronSpec = "* * * * */5"
              + image    = "my-awesome-cron-image"
            }
        }
      + object   = {
          + apiVersion = "stable.example.com/v1"
          + kind       = "CronTab"
          + metadata   = {
              # ...
            }
          + spec       = {
              + cronSpec = "* * * * */5"
              + image    = "my-awesome-cron-image"
            }
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
```

```
  Enter a value: yes

kubernetes_manifest.my_new_crontab: Creating...
kubernetes_manifest.my_new_crontab: Creation complete after 0s

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Confirm that Terraform created the custom resource.

```
$ kubectl get crontabs
NAME                 AGE
my-new-cron-object   5m37s
```

View the new custom resource.

```
$ kubectl describe crontab my-new-cron-object
Name:         my-new-cron-object
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  stable.example.com/v1
Kind:         CronTab
Metadata:
  Creation Timestamp:  2022-04-11T16:07:40Z
  Generation:          1
  Managed Fields:
    API Version:  stable.example.com/v1
    Fields Type:  FieldsV1
    fieldsV1:
      f:spec:
        f:cronSpec:
        f:image:
    Manager:         Terraform
    Operation:       Apply
    Time:            2022-04-11T16:07:40Z
  Resource Version:  2432053
  UID:               6dd859fc-8665-44ae-91f7-959cff8712b1
Spec:
  Cron Spec:  * * * * */5
  Image:      my-awesome-cron-image
Events:       <none>
```

Clean up your workspace

Destroy any resources you created once you're done with this tutorial.

Running `terraform destroy` will de-provision the NGINX deployment and service you created in this tutorial. Confirm your destroy with a `yes`.

```
$ terraform destroy
kubernetes_deployment.nginx: Refreshing state... [id=default/scalable-nginx-exampl
kubernetes_service.nginx: Refreshing state... [id=default/nginx-example]

## ...

Plan: 0 to add, 0 to change, 2 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

kubernetes_service.nginx: Destroying... [id=default/nginx-example]
kubernetes_service.nginx: Destruction complete after 0s
kubernetes_deployment.nginx: Destroying... [id=default/scalable-nginx-example]
kubernetes_deployment.nginx: Destruction complete after 0s

Destroy complete! Resources: 2 destroyed.
```

If you are using a kind Kubernetes cluster, run the following command to delete it.

```
$ kind delete cluster --name terraform-learn
```

If you followed a previous tutorial to set up a Kubernetes cluster, refer to the "Cleaning up your workspace" section of the tutorial to remove those resources as well. Otherwise, your Kubernetes cluster will remain running.

Next steps

In this tutorial, you configured the Terraform Kubernetes provider and used it to schedule, expose and scale an NGINX instance. You also used Terraform to create a custom resource definition and manage a custom resource.

To discover additional capabilities, visit the Terraform Kubernetes Provider Registry

For a more in-depth Kubernetes examples, complete the Deploy Consul and Vault on a Kubernetes Cluster using Run Triggers (runs on Google Cloud Platform) and Manage Kubernetes Custom Resources tutorials.

Was this tutorial helpful? Yes No

Previous

Next

This tutorial also appears in:

16 tutorials

Use Cases for Terraform

Use Terraform to perform common operations with other technologies, including Consul, Vault, Packer, and Kubernetes.