Refer: <https://medium.com/@Joachim8675309/deploy-kubernetes-apps-w-terraform-266f3e8028d2>

[**Terraform**](https://www.terraform.io/) can provision resources on [**Kubernetes**](https://kubernetes.io/), much in the same way you can provision cloud resources like [**AWS Cloud**](https://aws.amazon.com/), [**Google Cloud**](https://cloud.google.com/), and [**Azure**](https://azure.microsoft.com/).

This article demonstrates how to use [**Kubernetes Provider**](https://www.terraform.io/docs/providers/kubernetes/index.html)to deploy a service (using Helm Tiller as the example) on [**Amazon EKS**](https://aws.amazon.com/eks/). One advantage to using [**Amazon EKS**](https://aws.amazon.com/eks/) is that we can use the [**AWS provider**](https://www.terraform.io/docs/providers/aws/index.html) to fetch credentials necessary access our [**Kubernetes**](https://kubernetes.io/) cluster.

**Why use Terraform?**

The most obvious question would be why would you want to do this? The kubectl tool works fine and [**Kubernetes manifests**](https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/) ([**YAML**](https://yaml.org/)), so why not just stick with that?

There are a few situations where you need to orchestrate provisioning of resources on both the cloud provider and Kubernetes. I encountered at least two use cases:

* Standing up an [**Amazon EKS**](https://aws.amazon.com/eks/) cluster with some integration to AWS cloud resources installed into [**Kubernetes**](https://kubernetes.io/), e.g. external-dns for Rout53, tiller for helm charts, an ingress like nginx-ingress or aws-alb-ingress-controller, security with kube-iam, and so forth.
* Installing an application that is configured to use provisioned resources, like S3 buckets, SNS, SQS, ECR, IAM User, etc.

Additionally, you may want to use the templating that comes with [**Terraform**](https://www.terraform.io/) as an alternative to Helm charts.

**Related Article**

A while back (about 1.5 years), I wrote a similar article for doing this with [**GKE**](https://cloud.google.com/kubernetes-engine/) ([**Google Kubernetes Engine**](https://cloud.google.com/kubernetes-engine/)):

<https://medium.com/@Joachim8675309/deploy-kubernetes-apps-with-terraform-5b74e5891958>

**Required Tools**

You will need to install and configure the following tools:

* [**AWS CLI**](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html) needed to interact with AWS cloud resources. A profile with administrative access should be [**configured**](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html).
* [**EKSCtl**](https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html) (*eks-cuttle* or *exseey-cuttle*) tool to easily create an [**EKS**](https://aws.amazon.com/eks/) cluster. *[Note: this is optional should you want to eksctl to quickly create EKS cluster]*
* [**KubeCtl**](https://docs.aws.amazon.com/eks/latest/userguide/install-kubectl.html) (*koob-cuttle*), the [**Kubernetes**](https://kubernetes.io/) client tool, to interact with [**EKS**](https://aws.amazon.com/eks/)
* [**Helm**](https://docs.aws.amazon.com/eks/latest/userguide/helm.html) to install applications on a [**Kubernetes**](https://kubernetes.io/) cluster (Helm 2 explicitly, see below) *[Note: this is needed to demonstrate the Tiller service is working]*
* [**Terraform CLI**](https://www.terraform.io/downloads.html) to manage [**Kubernetes**](https://kubernetes.io/) and AWS resources, as well as create an EKS cluster.
* [**Bash Shell**](https://www.gnu.org/software/bash/) is not strictly required, but the commands in this tutorial are tested with bash *[Note: this is default in mac OS and popular Linux distros, or*[***msys2***](http://www.msys2.org/)*with Windows]*.

**Getting Helm 2**

You can get the download links for Helm2 tarball directly from GitHub:

* <https://github.com/helm/helm/releases/tag/v2.16.1>

Download and extract the tarball, so for example, if you downloaded the tarball for Mac OS, you could run (assuming bash shell):

**URL**=<https://get.helm.sh/helm-v2.16.1-darwin-amd64.tar.gz>  
**cd** ~/Downloads && **curl** **-sO** **$URL**  
**tar** xvzf **${URL##\*/}**  
**sudo cp** darwin-amd64/helm /usr/local/bin/helm2

We can test to see if it is in the path

**helm2** version | **cut** **-d**\" **-f**2

**Part 0: Setup Project Area**

In order to organize project files for scripts in this article, we’ll store them in eks-with-tiller directory:

**export PROJECT\_HOME**=**$HOME**/projects/eks-with-tiller  
**mkdir -p $PROJECT\_HOME** && **cd** **$PROJECT\_HOME**

**Part I: Creating a Kubernetes Cluster**

The easiest way to create an [**Amazon EKS**](https://aws.amazon.com/eks/) cluster is with the eksctl tool. Alternatively, you can use Terraform to stand up [**Amazon EKS**](https://aws.amazon.com/eks/) cluster as well.

The ultimate take-away form this is that you can install services and applications into [**Amazon EKS**](https://aws.amazon.com/eks/) created by any solution using the [**Kubernetes Provider**](https://www.terraform.io/docs/providers/kubernetes/index.html) in conjunction with [**AWS provider**](https://www.terraform.io/docs/providers/aws/index.html).

**Method 1: Using EKSCtl for EKS**

Run the following command:

**eksctl** create cluster \  
 **--name**=wonderful-unicorn \  
 **--kubeconfig**=wonderful-unicorn-kubeconfig.yaml

Note that the region will default to the region set in your current AWS profile in **$HOME**/.aws/config. For this tutorial, as an example, we’ll use us-west-2.

This process will take about 20 minutes. Once completed, test the results:

*# point KUBECONFIG to only our cluster*

**export** **KUBECONFIG**=**$PROJECT\_HOME**/wonderful-unicorn-kubeconfig.yaml

*# test kubectl works on new kubernetes cluster* **kubectl** get all **--all-namespaces**

**Method 2: Using Terraform for EKS**

To get started, we’ll create a small terraform file to describe our cluster:

**cat** <<-**CLUSTER\_EOF** > eks\_cluster.tf  
**variable** region {}  
**variable** eks\_cluster\_name {}

**module** "eks-cluster" {  
 source = "github.com/darkn3rd/eks-basic?ref=v0.0.1"   
 region = var.region   
 eks\_cluster\_name = var.eks\_cluster\_name   
}  
**CLUSTER\_EOF**

Next we’ll set some environment variables:

export TF\_VAR\_eks\_cluster\_name=wonderful\_unicorn  
export TF\_VAR\_region=us-west-2

Now we initialize our environment and apply the script:

**terraform** init  
**terraform** apply

This will take roughly 20 minutes, and afterward, we can test that the cluster works:

*# point KUBECONFIG to only our cluster* **export** **KUBECONFIG**=**$PROJECT\_HOME**/kubeconfig\_wonderful-unicorn

*# test kubectl works on new kubernetes cluster* **kubectl** get all **--all-namespaces**

**Part 2: Tiller Service Example**

Helm is the most popular package solution for Kubernetes and requires the Tiller service to install packages (at least with versions before Helm3).

**Setup**

We want to create the structure we’ll use for these code files that looks like this:

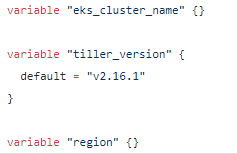
my\_modules/  
└── tiller/  
 ├── crb.tf  
 ├── deploy.tf  
 ├── provider.tf  
 ├── sa.tf  
 ├── svc.tf  
 └── variables.tf

With [**Bash shell**](https://www.gnu.org/software/bash/), you can create this using the following commands:

mkdir -p $PROJECT\_HOME/my\_modules/tiller/  
pushd $PROJECT\_HOME/my\_modules/tiller/  
touch ./{provider.tf,variables.tf,sa.tf,crb.tf,deploy.tf,svc.tf}  
popd

**Variables**

First we’ll edit the **tiller/variables.tf** that contains the variables we’ll use in this module.



**AWS and Kubernetes Providers**

Edit **tiller/provider.tf** with the following contents.

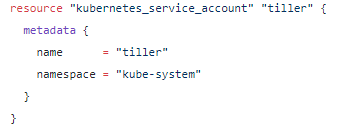


This script has two providers:

1. [**AWS provider**](https://www.terraform.io/docs/providers/aws/index.html): used to get the credentials from [**Amazon EKS**](https://aws.amazon.com/eks/) data sources.
2. [**Kubernetes Provider**](https://www.terraform.io/docs/providers/kubernetes/index.html): used to change state of resources in Kubernetes.

**Tiller Service Account Manifest**

Edit **tiller/sa.tf** with the following contents.



**Tiller Cluster Role Binding Manifest**

Edit **tiller/crb.tf** with the following contents.



**Tiller Deployment Manifest**

Edit **tiller/deploy.tf** with the following contents.

resource "kubernetes\_deployment" "tiller\_deploy" {

metadata {

name = "tiller-deploy"

namespace = "kube-system"

labels = {

app = "helm"

name = "tiller"

}

}

spec {

replicas = 1

selector {

match\_labels = {

app = "helm"

name = "tiller"

}

}

template {

metadata {

labels = {

app = "helm"

name = "tiller"

}

}

spec {

container {

name = "tiller"

image = "gcr.io/kubernetes-helm/tiller:${var.tiller\_version}"

port {

name = "tiller"

container\_port = 44134

}

port {

name = "http"

container\_port = 44135

}

env {

name = "TILLER\_NAMESPACE"

value = "kube-system"

}

env {

name = "TILLER\_HISTORY\_MAX"

value = "0"

}

liveness\_probe {

http\_get {

path = "/liveness"

port = "44135"

}

initial\_delay\_seconds = 1

timeout\_seconds = 1

}

readiness\_probe {

http\_get {

path = "/readiness"

port = "44135"

}

initial\_delay\_seconds = 1

timeout\_seconds = 1

}

image\_pull\_policy = "IfNotPresent"

}

service\_account\_name = "tiller"

automount\_service\_account\_token = true

}

}

}

depends\_on = [kubernetes\_service\_account.tiller]

}

**Tiller Service Manifest**

Edit **tiller/svc.tf** with the following contents.

As part of the Tiller setup, we create



**Part 3: Deploy the Service with Terraform**

We can either use the module directly or create a terraform script that references the module.

**Method 1: Directly Use the Module**

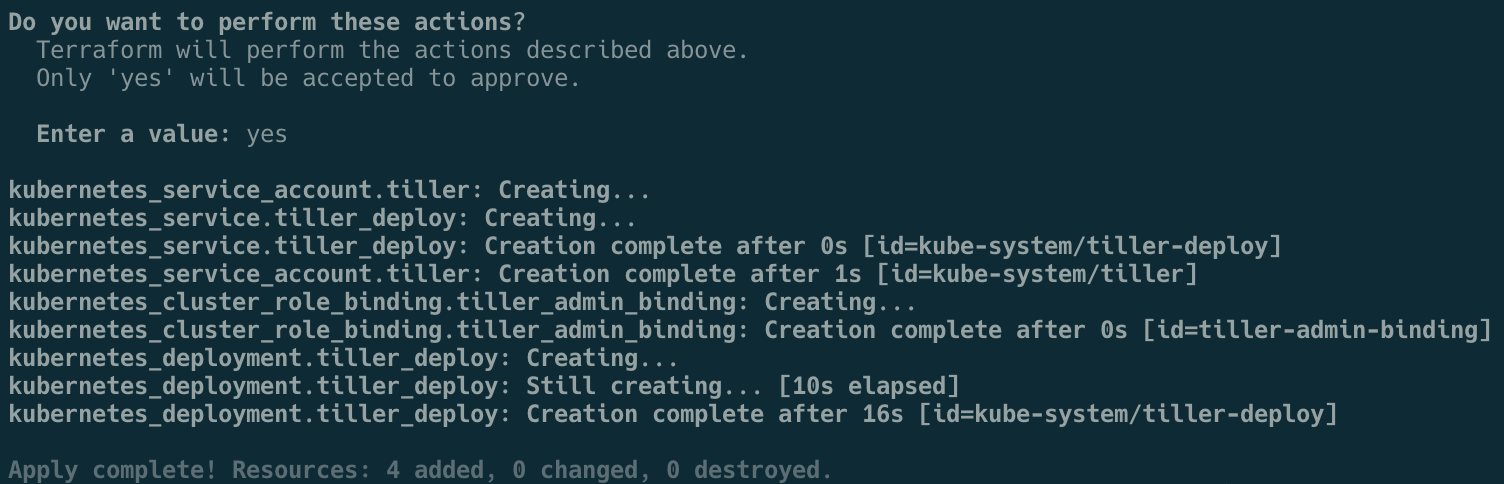
We can navigate to the module and initialize the dependencies with:

cd $PROJECT\_HOME/my\_modules/tiller && terraform init

Now we can run it directly by supplying the name of our EKS cluster and region (using us-west-2 as an example):

**terraform** apply \  
 **-var eks\_cluster\_name**=wonderful-unicorn  
 **-var region**=us-west-2

In the prompt, type yes, and you should see something like this:



**Output of Running Module**

**Method 2: Directly Use the Module**

Instead of running the code directly from the module, we create a [**Terraform**](https://www.terraform.io/) script that references the module:

cd **$PROJECT\_HOME**

*# create k8s\_addons.tf*  
cat <<-**K8SADDONS** > k8s\_addons.tf  
**module** "tiller\_install" {  
 **source** = "~/projects/eks-with-tiller/my\_modules/tiller"  
 **region** = "us-west-2"  
 **eks\_cluster\_name** = "wonderful-unicorn"  
}  
**K8SADDONS**

Change the values above to whatever is appropriate for the region. After, we can initialize the module dependencies and run this script:

terraform init  
terraform apply -target=tiller\_install

**Part 4: Testing the Deployed Service**

**Testing the Tiller Service**

If you have Helm 2 installed already, you can install a Helm chart package.

For example, you can install an Spinnaker Continuous Delivery solution (this takes about 4–8 minutes to install) with this:

**helm2** install stable/spinnaker

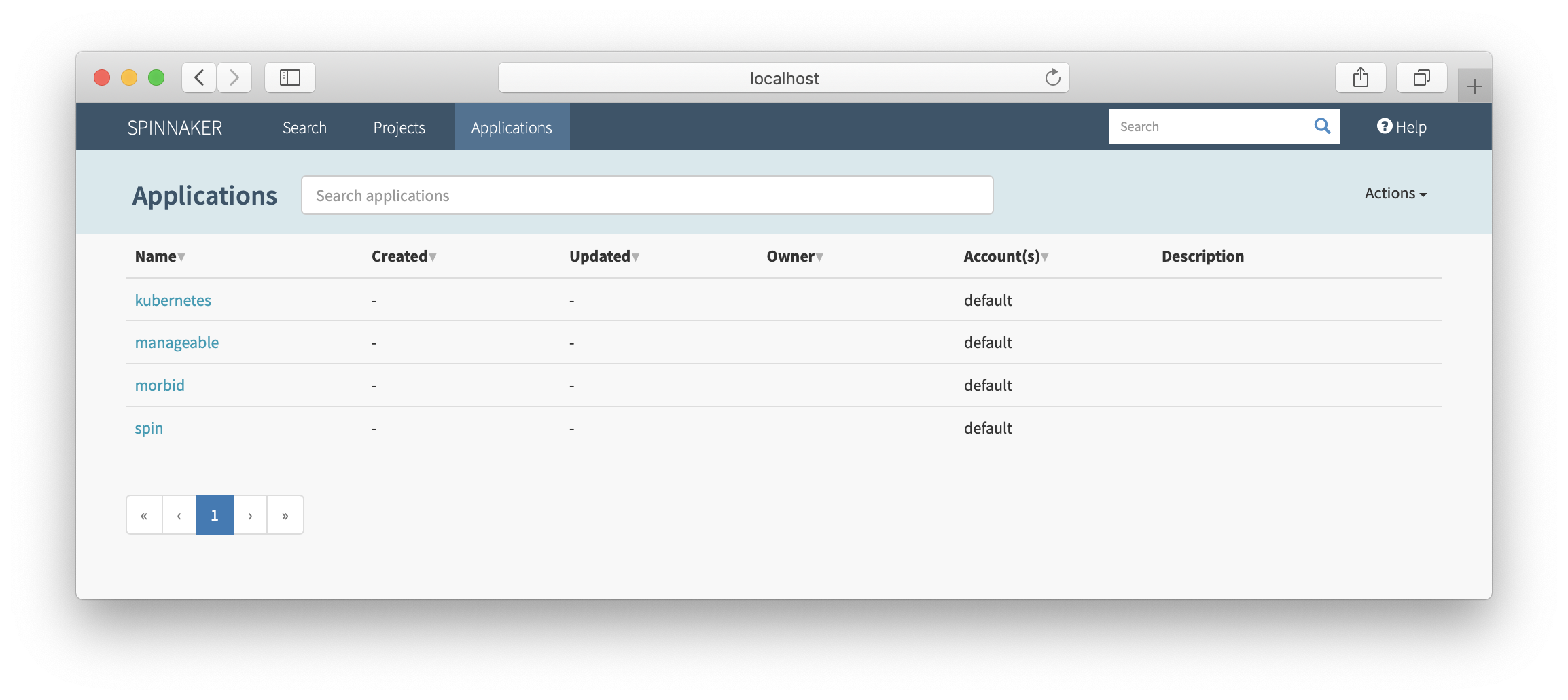
**export** **DECK\_POD**=$(**kubectl** get pods \  
 **--namespace** default \  
 **-l** "cluster=spin-deck" \  
 **-o** jsonpath="{.items[0].metadata.name}")

**export** **GATE\_POD**=$(**kubectl** get pods \  
 **--namespace** default \  
 **-l** "cluster=spin-gate" \  
 **-o** jsonpath="{.items[0].metadata.name}"  
)

**kubectl** port-forward **--namespace** default $GATE\_POD 8084 **&**

**kubectl** port-forward **--namespace** default $DECK\_POD 9000 **&**

Then try [http://localhost:9000:](http://localhost:9000/)



**Spinnaker Deck UI application**

**Part 5: Cleanup Resources**

**Delete Installed Chart**

For any helm charts, you can delete it with, with helm delete with the name that was auto-generated:

helm2 delete $(helm2 ls | grep spinnaker | cut -f1)

**Remove the Tiller Service**

We can remove Tiller with [**Terraform**](https://www.terraform.io/)

terraform destroy -var cluster=wonderful-unicorn

**Destroy the EKS Cluster**

The EKS cluster we created can be destroyed with the same tool used to create the cluster.

If you created the cluster with eksctl, you can run the following:

eksctl delete cluster --name wonderful-unicorn

If you used Terraform, you can do the following:

cd $PROJECT\_HOME  
terraform destroy

**Conclusion**

I hope this is useful tool to add for [**Terraform**](https://www.terraform.io/) and [**Kubernetes**](https://kubernetes.io/) journeys where they may intersect. This particular article leverages uses Amazon EKS, but really you could use other Kubernetes implementations provided that there is a data source to get the credentials to manage the cluster. Otherwise, you need to supply the credentials in another method, such as a **[kubeconfig](https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/" \t "_blank)** file.

Before I leave, one quick mention, is a neat tool called k2tf (<https://github.com/sl1pm4t/k2tf>) which can convert existing [**Kubernetes**](https://kubernetes.io/) manifests to [**Terraform**](https://www.terraform.io/) equivalents. Sometimes, [**Terraform**](https://www.terraform.io/) documentation may not be all that clear, especially with scarce examples, so the k2tf tool can help you get started.

Should you wish to use [**Terraform**](https://www.terraform.io/) as an alternative templating system to Helm, you could extract out the values with either helm install <package> --dry-run --debug or helm template <path>. This will show the final rendered [**Kubernetes**](https://kubernetes.io/)manifests, which you could then quickly convert with k2tf.

**Update**

I added material for standing up a cluster completely in [**Terraform**](https://www.terraform.io/) at the request of users from Reddit. Thanks for the input.