Kevin Marks

Provisioning Kubernetes clusters on AWS with Terraform and EKS

PUBLISHED 25 OCTOBER 2018

TL;DR: *In this guide, you will learn how to create clusters on the AWS Elastic Kubernetes Service (EKS) with eksctl and Terraform. By the end of the tutorial, you will automate creating three clusters (dev, staging, prod) complete with the AL2 Ingress Controller in a single click.*

EKS is a managed Kubernetes service, which means that Amazon Web Services (AWS) is fully responsible for managing the control plane.

In particular, AWS:

- Manages Kubernetes API servers and the etcd database
- Runs the Kubernetes control plane across three availability zones.
- Scales the control plane as you add more nodes to your cluster.
- Provides a mechanism to upgrade your control plane to a newer version.
- Rotates certificates.
- And more.

If you're running your clusters, you should still build all of these features.

However, when you use EKS, you outsource them to Amazon Web Service for a price: USD0.10 per hour per cluster.

> Please notice that Amazon Web Services has a 12 months free tier offering which you sign up for a new account. However, EKS is not part of the promotion.

The rest of the guide assumes that you have an account on Amazon Web Service.

If you don't, you can join us here.

This is a hands-on guide — *if you prefer to look at the code, you can do so here.*

Table of contents

1. Three popular options to provision an EKS cluster
2. But first, let's set up the AWS account
3. Eksctl: the quickest way to provision an EKS cluster
4. You can also define eksctl clusters in YAML, but the tools has its limits
5. You can provision an EKS cluster with Terraform too
6. Eksctl on Terraform — pros and cons
7. Testing the clusters by deploying a simple Hello World app
8. Routing traffic into the cluster with the ALB Ingress Controller
9. Deploying the Kubernetes Dashboard comes with few more provisions
10. Fully automated Dev, Staging, Production environments with Terraform modules
11. Summary and next steps

Three popular options to provision an EKS cluster

There are three popular options to run and deploy an EKS cluster:

1. You can create the cluster from the AWS web interface.
2. You can use the eksctl command-line utility.
3. You can define the cluster, its configuration, and its infrastructure in a text such as Terraform.

Each of these options **will** get you a working **cluster using the AWS interface is discouraged** and for a good reason.

There are plenty of configuration options and scenes that you have to complete before you can use the cluster.

When you create the cluster manually, can you be sure that:

- You did not forget to change one of the parameters?
- You can repeat precisely the same steps while creating a cluster for other environments?
- If you need to change, you can apply the same modifications in sequence to all clusters without any mistakes?

The process is error-prone and doesn't scale well if you have more than a single cluster.

A better option is to define a file that contains all the configuration flags and use that as a blueprint to create the cluster.

And that's precisely what you can do with tools such as eksctl and Terraform.

But first, let's set up the AWS account

Before you can start using eksctl and Terraform, you have to install the AWS CLI.

This tool is necessary to authenticate your requests to your account on Amazon Web Services.

You can find the official documentation on how to install the AWS CLI here.

After you install the AWS CLI you should run:

```
$ aws --version
aws-cli/1.0.0 Python/3.6.1 Darwin/18.0.0 botocore/0.34
```

If you can see the version in the output, that means the installation is successful.

Next, you need to link your account to the AWS CLI.

For this part, you will need:

1. AWS Access Key ID,
2. AWS Secret Access Key,
3. Default region name,
4. Default output format.

The *essential* parts you need are the first two: the **Access Key ID** and the **Secret Access Key**.

These credentials are displayed only once after you create a user on the AWS web interface.

To do so, follow these instructions:

Log in to your AWS Management Console.

Now that you have the keys, you enter all the details.

```
$ aws configure
AWS Access Key ID [None]: <enter the access key>
AWS Secret Access Key [None]: <enter the secret key>
Default region name [None]: <enter region>
Default output format [None]: <enter>
$
```

> Please notice that the list of available regions can be found here

The AWS CLI lets you interact with AWS without using the web interface.

You can try listing all your EKS clusters with:

```
$ aws eks list-clusters
{
    "clusters": []
}
$
```

An empty list — it makes sense, you haven't created any yet.

Now that you have your account on AWS set up, it's time to use eksctl.

Eksctl: the quickest way to provision an EKS cluster

Eksctl is a convenient command-line tool to create an EKS cluster with a few simple commands.

So what's the difference with the AWS CLI?

Isn't the AWS CLI enough to create resources?

The AWS CLI has a command to create an EKS cluster: `aws eks create-cluster`.

However, the command only creates a control plane.

It does not create any worker node, set up the authentication, permissions, etc.

On the other hand, eksctl is an `aws eks` on steroids.

With a single command, you have a fully functioning cluster.

Before diving into an example, you should install the eksctl binary.

You can find the instructions on how to install eksctl from the official project page.

You can verify that eksctl is installed correctly with:

```
$ eksctl version
0.1.0
$
```

Eksctl uses the credentials from the AWS CLI to connect to your account.

So you can spin up a cluster with:

```
$ eksctl create cluster
```

The command makes a few assumptions about the cluster that you want:

- It creates a control plane (managed by AWS).
- It plots two workers nodes.
- It selects the `m5.large` as an instance type.
- It creates the cluster in the `us-west-2` region.

If the cluster isn't quite what you had in mind, you can easily customize the settings to fit your needs.

Of course, you can change the region or include an instance type that is covered by the free offer such as a `t2.micro`.

Also, why not including autoscaling?

You can create a new cluster with:

```
$ eksctl create cluster \
    --name test-cluster \
    --node-type t2.micro \
    --nodes 3 \
    --nodes-min 3 \
    --nodes-max 5 \
    --region us-central-1
```

The command creates a control plane:

- In the region `us-central-1` (Frankfurt)
- With cluster name as "test-kits-cluster".
- Using `t2.micro` instances.
- The worker instances will autoscale based on load (from 3 to a maximum of 5 nodes).

> Please notice that the command could take about 15 to 20 minutes to complete.

In this time, eksctl:

1. Uses the AWS CLI credentials to connect to your account on AWS.
2. Creates the required templates for the EC2 stacks as well as the node groups. That includes AWS images, security groups, etc.
3. Creates all the necessary networking plumbing such as the VPC, subnets, and IP addresses.
4. Generates the credentials needed to access the Kubernetes cluster — the kubeconfig.

While you are waiting for the cluster to be provisioned, you should download kubectl — the command-line tool to connect and manage the Kubernetes cluster.

Kubectl can be downloaded from here.

You can check that the binary is installed successfully with:

```
$ kubectl version --client
```

It should give you the version as the output.

Once your cluster is ready, you will be greeted by the following line.

```
Output truncated
[✓]  EKS cluster "test-kits-cluster" in "us-central-1" region is ready
$
```

You can verify that the cluster is running by using:

```
$ eksctl get cluster --name test-kits-cluster --region us-central-1 [-o]
```

It works!

Let's list all the Pods in the cluster:

```
$ kubectl get pods --all-namespaces
kube-system   aws-node-24c8q                  1/1     Running   0          3m22s
kube-system   aws-node-cbhbb                  1/1     Running   0          3m20s
kube-system   aws-node-cdfbf                  1/1     Running   0          3m18s
kube-system   coredns-5576d9f79-7ldwg         1/1     Running   0          20m
kube-system   coredns-5576d9f79-ttqqq         1/1     Running   0          20m
kube-system   kube-proxy-ds4qf                1/1     Running   0          3m18s
kube-system   kube-proxy-dtbbf                1/1     Running   0          3m20s
kube-system   kube-proxy-gkjzm                1/1     Running   0          3m22s
$
```

You can see from the `kube-system` namespace, that Kubernetes created the mandatory pods needed to run the cluster.

With this, **you have successfully created and connected to a fully functional Kubernetes cluster.**

Congrats!

Contain your excitement, though — you will immediately destroy the cluster.

There's a better way to create clusters with eksctl, and that's by defining what you want in a YAML file.

You can also define eksctl clusters in YAML, but the tools has its limits

When you have all the cluster configuration in a single file, you can:

1. Store in Git or any other version control.
2. Share it with your friends and colleagues.
3. Recall what the last configuration applied to the cluster was.

Before exploring the YAML configuration file eksctl, let's destroy the current cluster with:

```
$ eksctl delete cluster --name test-kits-cluster --region us-central-1 [-o]
```

Do not forget to let the command finish and do its job, otherwise terminating prematurely may leave a few dangling resources (which you will be billed for).

You should see the following command output after the deletion is completed:

```
Output truncated
[✓]  all cluster resources were deleted
$
```

Eksctl lets you create clusters that are defined in YAML format.

You can define your cluster and node specification and pass that file to eksctl so it can create the resources.

Let's have a look at how it works.

Create a `cluster.yaml` file with the following content:

```
# cluster.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: test-kits
  region: us-central-1
nodeGroups:
  - name: worker-group
    instanceType: t2.micro
    desiredCapacity: 3
    minSize: 3
    maxSize: 5
```

You can already guess what the above cluster will look like when deployed.

It's the cluster that you created earlier with the command line arguments, but this time all of the requirements are stored in the YAML.

It's a cluster:

- In the region `us-central-1` (Frankfurt).
- With cluster name as "test-kits-cluster".
- Using `t2.micro` instances.
- The worker instances will autoscale based on load (from 3 to a maximum of 5 nodes).

You can create a cluster by feeding the YAML configuration with:

```
$ eksctl create cluster -f cluster.yaml
```

Again, be patient here.

The command will need 15-15 minutes to complete provisioning the resources.

You can verify that the cluster was created successfully with:

```
$ eksctl get cluster --name test-kits
NAME    REGION  EKSCTL CONFIG
test-kits       True    ACTIVE
$
```

And you are done, you've successfully provisioned a cluster with eksctl and YAML.

If you want, you can save that file in version control.

You stop and yourself what happens when you apply the same configuration again?

Does it update the cluster?

Let's try:

```
$ eksctl create cluster -f cluster.yaml
```

```
$ kubectl get -n kube-system configmap/aws-auth
error: Error from server (NotFound): configmaps "aws-auth" not found
```

You won't be able to amend the specification since the cruder command is only used in the beginning to make the cluster.

At the moment, there is no command designed to read the YAML and update the cluster to the latest changes.

> Please notice that the feature is on the roadmap, though.

If you want to describe the cluster as a static file, but incrementally update the configuration, you might find Terraform more suitable.

You can provision an EKS cluster with Terraform too

Terraform is an open-source infrastructure as Code tool.

Instead of writing the code to create the infrastructure, you define a plan of what you want to be executed, and you let Terraform create the resources on your behalf.

The plan isn't written in YAML though.

Instead Terraform uses a bespoke called HCL - HashiCorp Configuration Language.

In other words, you use HCL to declare the infrastructure you want to be deployed, and Terraform executes the instructions.

Terraform uses plugins called providers to interface with the resources in the cloud provider.

This further expands with modules as a group of resources and are the building blocks that you are going to use to create a stack.

But let's take a break from the theory and see those concepts in practice.

Before you can create a cluster with Terraform, you should install the binary.

You can find the instructions how to install the Terraform CLI from the official documentation.

Verify that the Terraform tool has been installed correctly with:

```
$ terraform version
Terraform v0.12.5

$
```

Now start by creating a new folder, and in that folder create a file named `main.tf`.

The `.tf` extension is for Terraform files.

In the `main.tf` copy and paste the following code:

```
provider "aws" {
  region = "eu-west-1"
}

data "aws_eks_cluster" "cluster" {
  name = module.eks_cluster_id
}

data "aws_eks_cluster_auth" "cluster" {
  name = module.eks_cluster_id
}

provider "kubernetes" {
  host                   = data.aws_eks_cluster.cluster.endpoint
  cluster_ca_certificate = base64decode(data.aws_eks_cluster.cluster.certi
  token                  = data.aws_eks_cluster_auth.cluster.token
  load_config_file       = false
  version                = "~> 1.11"
}

data "aws_availability_zones" "available" {
}

locals {
  cluster_name = "my-cluster"
}

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.45.0"

  name                 = "eks-vpc"
  cidr                 = "172.16.0.0/16"
  azs                  = data.aws_availability_zones.available.names
  private_subnets      = ["172.16.1.0/24", "172.16.2.0/24", "172.16.3.0/24
  public_subnets       = ["172.16.4.0/24", "172.16.5.0/24", "172.16.6.0/24
  enable_nat_gateway   = true
  single_nat_gateway   = true
  enable_dns_hostnames = true

  public_subnet_tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/elb"                      = "1"
  }

  private_subnet_tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/internal-elb"             = "1"
  }

}

module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "12.2.0"

  cluster_name  = "${local.cluster_name}"
  cluster_version = "1.13"
  subnets         = module.vpc.private_subnets

  tags = {
    Environment = "test"
    GitHash     = "${var.circle_sha}"
  }

  vpc_id = module.vpc.vpc_id

  node_groups = {
    first = {
      desired_capacity = 3
      max_capacity     = 10
      min_capacity     = 3

      instance_type = "m5.large"
      k2 = "${var.circle_sha}"
    }
  }

  write_kubeconfig   = true
  config_output_path = "./"
}
```

> You can find the code in this GitHub repository too.

That's a lot of code!

Bear in mind with me.

You will learn everything about it as soon as you're done creating the cluster.

In the same folder, run:

```
$ terraform init
```

The command will initialise Terraform and create two more folders as well as a state file.

The state file is used to keep track of the resources that have been created already.

Consider this as a checkpoint, without it Terraform won't know what has been already created or updated.

There is another command that you can utilize in your understanding with Terraform.

To quickly check if the configuration doesn't have any configuration errors, you can do so with:

```
$ terraform validate
Success! The configuration is valid.

$
```

Next, you should run:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will
be persisted to local or remote state storage.

data.aws_availability_zones.available: Refreshing state...
data.aws_eks_cluster.cluster: Refreshing state...
data.aws_eks_cluster_auth.cluster: Refreshing state...

------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.eks.aws_cloudwatch_log_group.this[0] will be created
  + resource "aws_cloudwatch_log_group" "this[0]"
      id          = (known after apply)
      arn         = (known after apply)
      name        = "/aws/eks/my-cluster/cluster"
      retention_in_days = 90
      tags        = (known after apply)

  # module.eks.aws_iam_policy.cluster_elb_sl_role_creation[0] will be c
  + resource "aws_iam_policy" "cluster_elb_sl_role_creation[0]"
      arn         = (known after apply)
      description = "Cluster ELB Service Linked Role creation for kubelets"
      name        = (known after apply)
      name_prefix = "eks-cluster-elb-sl-role-creation-20190729085338669
      policy      = (known after apply)

  # module.eks.aws_iam_policy.kubelet_log_4_cw[0] will be created
  + resource "aws_iam_policy" "kubelet_log_4_cw[0]"
      arn         = (known after apply)
      description = "EKS kubelet node CloudWatch Logs policy for cluster
      name        = (known after apply)
      name_prefix = "eks-kubelet-log-4-cw20190729085338669000001"
      policy      = (known after apply)

  # module.eks.aws_iam_role.cluster[0] will be created
  + resource "aws_iam_role" "cluster[0]"
      arn           = (known after apply)
      assume_role_policy = <<EOF
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "eks.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      EOF
      create_date   = (known after apply)
      force_detach_policies = false
      name          = "eks-cluster-20190729085338669000001"
      path          = "/"
      unique_id     = (known after apply)

  # module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSClusterPo
  + resource "aws_iam_role_policy_attachment" "cluster_AmazonEKSClusterPo
      id          = (known after apply)
      role        = "eks-cluster-20190729085338669000001"
      policy_arn  = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"

  # module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSServicePo
  + resource "aws_iam_role_policy_attachment" "cluster_AmazonEKSServicePo
      id          = (known after apply)
      role        = "eks-cluster-20190729085338669000001"
      policy_arn  = "arn:aws:iam::aws:policy/AmazonEKSServicePolicy"

  # module.eks.aws_security_group.cluster[0] will be created
  + resource "aws_security_group" "cluster[0]"
      arn         = (known after apply)
      description = "EKS cluster security group."
      name        = (known after apply)
      name_prefix = "eks-cluster-20190729085338669000001"
      owner_id    = (known after apply)
      tags        = (known after apply)
      vpc_id      = "vpc-0ff0a0a0a0a0a0a0a"

  # module.eks.aws_security_group_rule.cluster_egress_internet[0] will
  + resource "aws_security_group_rule" "cluster_egress_internet[0]"
      id          = (known after apply)
      type        = "egress"
      cidr_blocks = [
        "0.0.0.0/0",
      ]
      description = "Allow cluster egress access to the Internet."
      from_port   = 0
      protocol    = "-1"
      security_group_id = (known after apply)
      self      = false
      to_port   = 0

  # module.eks.aws_security_group_rule.cluster_https_ingress[0] will be
  + resource "aws_security_group_rule" "cluster_https_ingress[0]"
      id          = (known after apply)
      type        = "ingress"
      cidr_blocks = [
        "172.16.0.0/16",
      ]
      description = "Allow pods to communicate with the cluster API Ser
      from_port   = 443
      protocol    = "tcp"
      security_group_id = (known after apply)
      self      = false
      to_port   = 443

  # module.eks.aws_security_group_rule.kubelet_log_4_cw[0] will be crea
  + resource "aws_security_group_rule" "kubelet_log_4_cw[0]"
      id          = (known after apply)
      type        = "egress"
      description = "Allow kubelet node CloudWatch Logs access"
      from_port   = 443
      protocol    = "tcp"
      security_group_id = (known after apply)
      self      = false
      to_port   = 443

Plan: 14 to add, 0 to change, 0 to destroy.

$
```

The command will initialise Terraform and create two more folders as well as a state file.

The state file is used to keep track of the resources that have been created already.

Consider this as a checkpoint, without it Terraform won't know what has been already created or updated.

There is another command that you can utilize in your understanding with Terraform.

To quickly check if the configuration doesn't have any configuration errors, you can do so with:

```
$ terraform validate
Success! The configuration is valid.

$
```

Next, you should run:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will
be persisted to local or remote state storage.

data.aws_availability_zones.available: Refreshing state...
data.aws_eks_cluster.cluster: Refreshing state...
data.aws_eks_cluster_auth.cluster: Refreshing state...

------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.eks.aws_cloudwatch_log_group.this[0] will be created
  + resource "aws_cloudwatch_log_group" "this[0]"
      id          = (known after apply)
      arn         = (known after apply)
      name        = "/aws/eks/my-cluster/cluster"
      retention_in_days = 90
      tags        = (known after apply)

  # module.eks.aws_iam_policy.cluster_elb_sl_role_creation[0] will be c
  + resource "aws_iam_policy" "cluster_elb_sl_role_creation[0]"
      arn         = (known after apply)
      description = "Cluster ELB Service Linked Role creation for kubelets"
      name        = (known after apply)
      name_prefix = "eks-cluster-elb-sl-role-creation-20190729085338669
      policy      = (known after apply)

  # module.eks.aws_iam_policy.kubelet_log_4_cw[0] will be created
  + resource "aws_iam_policy" "kubelet_log_4_cw[0]"
      arn         = (known after apply)
      description = "EKS kubelet node CloudWatch Logs policy for cluster
      name        = (known after apply)
      name_prefix = "eks-kubelet-log-4-cw20190729085338669000001"
      policy      = (known after apply)

  # module.eks.aws_iam_role.cluster[0] will be created
  + resource "aws_iam_role" "cluster[0]"
      arn           = (known after apply)
      assume_role_policy = <<EOF
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "eks.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      EOF
      create_date   = (known after apply)
      force_detach_policies = false
      name          = "eks-cluster-20190729085338669000001"
      path          = "/"
      unique_id     = (known after apply)

  # module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSClusterPo
  + resource "aws_iam_role_policy_attachment" "cluster_AmazonEKSClusterPo
      id          = (known after apply)
      role        = "eks-cluster-20190729085338669000001"
      policy_arn  = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"

  # module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSServicePo
  + resource "aws_iam_role_policy_attachment" "cluster_AmazonEKSServicePo
      id          = (known after apply)
      role        = "eks-cluster-20190729085338669000001"
      policy_arn  = "arn:aws:iam::aws:policy/AmazonEKSServicePolicy"

  # module.eks.aws_security_group.cluster[0] will be created
  + resource "aws_security_group" "cluster[0]"
      arn         = (known after apply)
      description = "EKS cluster security group."
      name        = (known after apply)
      name_prefix = "eks-cluster-20190729085338669000001"
      owner_id    = (known after apply)
      tags        = (known after apply)
      vpc_id      = "vpc-0ff0a0a0a0a0a0a0a"

  # module.eks.aws_security_group_rule.cluster_egress_internet[0] will
  + resource "aws_security_group_rule" "cluster_egress_internet[0]"
      id          = (known after apply)
      type        = "egress"
      cidr_blocks = [
        "0.0.0.0/0",
      ]
      description = "Allow cluster egress access to the Internet."
      from_port   = 0
      protocol    = "-1"
      security_group_id = (known after apply)
      self      = false
      to_port   = 0

  # module.eks.aws_security_group_rule.cluster_https_ingress[0] will be
  + resource "aws_security_group_rule" "cluster_https_ingress[0]"
      id          = (known after apply)
      type        = "ingress"
      cidr_blocks = [
        "172.16.0.0/16",
      ]
      description = "Allow pods to communicate with the cluster API Ser
      from_port   = 443
      protocol    = "tcp"
      security_group_id = (known after apply)
      self      = false
      to_port   = 443

  # module.eks.aws_security_group_rule.kubelet_log_4_cw[0] will be crea
  + resource "aws_security_group_rule" "kubelet_log_4_cw[0]"
      id          = (known after apply)
      type        = "egress"
      description = "Allow kubelet node CloudWatch Logs access"
      from_port   = 443
      protocol    = "tcp"
      security_group_id = (known after apply)
      self      = false
      to_port   = 443

Plan: 14 to add, 0 to change, 0 to destroy.

$
```

The export is valid only for the current terminal session.

Now that you've created the cluster, it's time to go back and discuss the Terraform file.

The Terraform file that you just executed is divided into two blocks:

```
provider "aws" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.47.0"

  name                 = "eks-vpc"
  cidr                 = "172.16.0.0/16"
  azs                  = ["172.16.0.0/16", "172.16.1.0/24", "172.16.2.0/24
  private_subnets      = ["172.16.3.0/24", "172.16.4.0/24", "172.16.5.0/24
  public_subnets       = ["172.16.6.0/24", "172.16.7.0/24", "172.16.8.0/24

  # Instances
}

module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "12.2.0"

  cluster_name    = "${local.cluster_name}"
  cluster_version = "1.17"
  subnets         = module.vpc.private_subnets

  vpc_id = module.vpc.vpc_id

  # Instances
}
```

The first block is the VPC module.

In this part, you instruct Terraform to create:

- A VPC.
- Three private and three public subnets.
- A single NAT gateway.
- Tags for the subnets.

The tags for subnets are quite crucial as these are used by AWS to automatically provision public and internal load balancers in the appropriate subnets.

The second important block in the Terraform file is the EKS cluster module:

```
provider "aws" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.47.0"

  name                 = "eks-vpc"
  cidr                 = "172.16.0.0/16"
  azs                  = ["172.16.0.0/16", "172.16.1.0/24", "172.16.2.0/24
  private_subnets      = ["172.16.3.0/24", "172.16.4.0/24", "172.16.5.0/24
  public_subnets       = ["172.16.6.0/24", "172.16.7.0/24", "172.16.8.0/24

  # Instances
}

module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "12.2.0"

  cluster_name    = "${local.cluster_name}"
  cluster_version = "1.17"
  subnets         = module.vpc.private_subnets

  vpc_id = module.vpc.vpc_id

  # Instances
}
```

The EKS module is in charge of:

- Creating the control plane.
- Setting up autoscaling groups.
- Setting up the proper security groups.
- Creating the kubeconfig file.
- And more.

Notice how the EKS cluster has to be created into a VPC.

Also, the worker nodes for your Kubernetes cluster should be deployed in the private subnets.

You can define those two constraints with:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.47.0"

  # Instances
}

module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "12.2.0"

  # Instances

  cluster_name    = "${local.cluster_name}"
  cluster_version = "1.17"
  subnets         = module.vpc.private_subnets

  vpc_id = module.vpc.vpc_id

  # Instances
}
```

The last part of the Terraform file is the following:

```
data "aws_eks_cluster" "cluster" {
  name = module.eks.cluster_id
}

data "aws_eks_cluster_auth" "cluster" {
  name = module.eks.cluster_id
}

provider "kubernetes" {
  host                   = data.aws_eks_cluster.cluster.endpoint
  cluster_ca_certificate = base64decode(data.aws_eks_cluster.certi
  token                  = data.aws_eks_cluster_auth.cluster.token
  load_config_file       = false
  version                = "~> 1.11"
}
```

The code is necessary to:

- Set up the kmb permissions to connect to the cluster.
- Pull the cluster to make sure it's ready.

And that's all!

Eksctl vs Terraform — pros and cons

You can already tell the main differences between eksctl and Terraform:

- Both create an EKS cluster.
- Both expect a valid kubeconfig file.
- The configuration for eksctl is more concise.
- Terraform is more granular. You have to craft every single resource carefully.

So which one should you use?

For small experiments, you should consider eksctl.

With a short command you can quickly create a cluster.

For production infrastructure where you want to configure every single detail of your cluster, you should consider using Terraform.

But there's another leaderboard where you should pick Terraform, and that's incremental updates.

Let's imagine that you want to add a second pool of server to your cluster.

Perhaps you want to add GPU nodes to your cluster so that you can train your machine learning models.

You can amend the file as follows:

```
provider "aws" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.47.0"

  # Instances
}

module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "12.2.0"

  # Instances

  cluster_name    = "${local.cluster_name}"
  cluster_version = "1.17"
  subnets         = module.vpc.private_subnets

  vpc_id = module.vpc.vpc_id

  node_groups = {
    first = {
      desired_capacity = 3
      max_capacity     = 10
      min_capacity     = 3

      instance_type = "m5.large"
      k2 = "${var.circle_sha}"
    }

    gpu = {
      desired_capacity = 1
      max_capacity     = 10
      min_capacity     = 1

      instance_type = "p3.2xlarge"
      k2 = "${var.circle_sha}"
    }
  }

  write_kubeconfig   = true
  config_output_path = "./"
}
```

> You can find the amended code in this GitHub repository.

You can dry run the changes with:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will
be persisted to local or remote state storage.

data.aws_availability_zones.available: Refreshing state...
data.aws_eks_cluster.cluster: Refreshing state...
data.aws_eks_cluster_auth.cluster: Refreshing state...

------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.eks.aws_cloudwatch_log_group.this[0] will be created
  + resource "aws_cloudwatch_log_group" "this[0]"
      id          = (known after apply)
      arn         = (known after apply)
      name        = "/aws/eks/my-cluster/cluster"
      retention_in_days = 90
      tags        = (known after apply)

  # module.eks.aws_iam_policy.cluster_elb_sl_role_creation[0] will be c
  + resource "aws_iam_policy" "cluster_elb_sl_role_creation[0]"
      arn         = (known after apply)
      description = "Cluster ELB Service Linked Role creation for kubelets"
      name        = (known after apply)
      name_prefix = "eks-cluster-elb-sl-role-creation-20190729085338669
      policy      = (known after apply)

  # module.eks.aws_iam_policy.kubelet_log_4_cw[0] will be created
  + resource "aws_iam_policy" "kubelet_log_4_cw[0]"
      arn         = (known after apply)
      description = "EKS kubelet node CloudWatch Logs policy for cluster
      name        = (known after apply)
      name_prefix = "eks-kubelet-log-4-cw20190729085338669000001"
      policy      = (known after apply)

  # module.eks.aws_iam_role.cluster[0] will be created
  + resource "aws_iam_role" "cluster[0]"
      arn           = (known after apply)
      assume_role_policy = <<EOF
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "eks.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      EOF
      create_date   = (known after apply)
      force_detach_policies = false
      name          = "eks-cluster-20190729085338669000001"
      path          = "/"
      unique_id     = (known after apply)

  # module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSClusterPo
  + resource "aws_iam_role_policy_attachment" "cluster_AmazonEKSClusterPo
      id          = (known after apply)
      role        = "eks-cluster-20190729085338669000001"
      policy_arn  = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"

  # module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSServicePo
  + resource "aws_iam_role_policy_attachment" "cluster_AmazonEKSServicePo
      id          = (known after apply)
      role        = "eks-cluster-20190729085338669000001"
      policy_arn  = "arn:aws:iam::aws:policy/AmazonEKSServicePolicy"

  # module.eks.aws_security_group.cluster[0] will be created
  + resource "aws_security_group" "cluster[0]"
      arn         = (known after apply)
      description = "EKS cluster security group."
      name        = (known after apply)
      name_prefix = "eks-cluster-20190729085338669000001"
      owner_id    = (known after apply)
      tags        = (known after apply)
      vpc_id      = "vpc-0ff0a0a0a0a0a0a0a"

  # module.eks.aws_security_group_rule.cluster_egress_internet[0] will
  + resource "aws_security_group_rule" "cluster_egress_internet[0]"
      id          = (known after apply)
      type        = "egress"
      cidr_blocks = [
        "0.0.0.0/0",
      ]
      description = "Allow cluster egress access to the Internet."
      from_port   = 0
      protocol    = "-1"
      security_group_id = (known after apply)
      self      = false
      to_port   = 0

  # module.eks.aws_security_group_rule.cluster_https_ingress[0] will be
  + resource "aws_security_group_rule" "cluster_https_ingress[0]"
      id          = (known after apply)
      type        = "ingress"
      cidr_blocks = [
        "172.16.0.0/16",
      ]
      description = "Allow pods to communicate with the cluster API Ser
      from_port   = 443
      protocol    = "tcp"
      security_group_id = (known after apply)
      self      = false
      to_port   = 443

  # module.eks.aws_security_group_rule.kubelet_log_4_cw[0] will be crea
  + resource "aws_security_group_rule" "kubelet_log_4_cw[0]"
      id          = (known after apply)
      type        = "egress"
      description = "Allow kubelet node CloudWatch Logs access"
      from_port   = 443
      protocol    = "tcp"
      security_group_id = (known after apply)
      self      = false
      to_port   = 443

Plan: 14 to add, 0 to change, 0 to destroy.

$
```

Once you're ready, you can apply the changes with:

```
$ terraform apply
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will
be persisted to local or remote state storage.

data.aws_availability_zones.available: Refreshing state...
data.aws_eks_cluster.cluster: Refreshing state...
data.aws_eks_cluster_auth.cluster: Refreshing state...

------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.eks.aws_cloudwatch_log_group.this[0] will be created
  + resource "aws_cloudwatch_log_group" "this[0]"
      id          = (known after apply)
      arn         = (known after apply)
      name        = "/aws/eks/my-cluster/cluster"
      retention_in_days = 90
      tags        = (known after apply)

  # module.eks.aws_iam_policy.cluster_elb_sl_role_creation[0] will be c
  + resource "aws_iam_policy" "cluster_elb_sl_role_creation[0]"
      arn         = (known after apply)
      description = "Cluster ELB Service Linked Role creation for kubelets"
      name        = (known after apply)
      name_prefix = "eks-cluster-elb-sl-role-creation-20190729085338669
      policy      = (known after apply)

  # module.eks.aws_iam_policy.kubelet_log_4_cw[0] will be created
  + resource "aws_iam_policy" "kubelet_log_4_cw[0]"
      arn         = (known after apply)
      description = "EKS kubelet node CloudWatch Logs policy for cluster
      name        = (known after apply)
      name_prefix = "eks-kubelet-log-4-cw20190729085338669000001"
      policy      = (known after apply)

  # module.eks.aws_iam_role.cluster[0] will be created
  + resource "aws_iam_role" "cluster[0]"
      arn           = (known after apply)
      assume_role_policy = <<EOF
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "eks.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      EOF
      create_date   = (known after apply)
      force_detach_policies = false
      name          = "eks-cluster-20190729085338669000001"
      path          = "/"
      unique_id     = (known after apply)

  # module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSClusterPo
  + resource "aws_iam_role_policy_attachment" "cluster_AmazonEKSClusterPo
      id          = (known after apply)
      role        = "eks-cluster-20190729085338669000001"
      policy_arn  = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"

  # module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSServicePo
  + resource "aws_iam_role_policy_attachment" "cluster_AmazonEKSServicePo
      id          = (known after apply)
      role        = "eks-cluster-20190729085338669000001"
      policy_arn  = "arn:aws:iam::aws:policy/AmazonEKSServicePolicy"

  # module.eks.aws_security_group.cluster[0] will be created
  + resource "aws_security_group" "cluster[0]"
      arn         = (known after apply)
      description = "EKS cluster security group."
      name        = (known after apply)
      name_prefix = "eks-cluster-20190729085338669000001"
      owner_id    = (known after apply)
      tags        = (known after apply)
      vpc_id      = "vpc-0ff0a0a0a0a0a0a0a"

  # module.eks.aws_security_group_rule.cluster_egress_internet[0] will
  + resource "aws_security_group_rule" "cluster_egress_internet[0]"
      id          = (known after apply)
      type        = "egress"
      cidr_blocks = [
        "0.0.0.0/0",
      ]
      description = "Allow cluster egress access to the Internet."
      from_port   = 0
      protocol    = "-1"
      security_group_id = (known after apply)
      self      = false
      to_port   = 0

  # module.eks.aws_security_group_rule.cluster_https_ingress[0] will be
  + resource "aws_security_group_rule" "cluster_https_ingress[0]"
      id          = (known after apply)
      type        = "ingress"
      cidr_blocks = [
        "172.16.0.0/16",
      ]
      description = "Allow pods to communicate with the cluster API Ser
      from_port   = 443
      protocol    = "tcp"
      security_group_id = (known after apply)
      self      = false
      to_port   = 443

  # module.eks.aws_security_group_rule.kubelet_log_4_cw[0] will be crea
  + resource "aws_security_group_rule" "kubelet_log_4_cw[0]"
      id          = (known after apply)
      type        = "egress"
      description = "Allow kubelet node CloudWatch Logs access"
      from_port   = 443
      protocol    = "tcp"
      security_group_id = (known after apply)
      self      = false
      to_port   = 443

Plan: 14 to add, 0 to change, 0 to destroy.

$
```

You've configured and updated the cluster, but still, you haven't deployed any application.

Let's create a simple deployment.

Testing the cluster by deploying a simple Hello World app

```
$ kubectl get -n kube-system configmap/aws-auth -o yaml
apiVersion: v1
data:
  mapRoles: |
    - rolearn: arn:aws:iam::000000000000:role/eks-cluster-20190729085338669
      username: system:node:{{EC2PrivateDNSName}}
      groups:
      - system:bootstrappers
      - system:nodes
kind: ConfigMap
metadata:
  creation_timestamp: 2019-07-29T08:54:12Z
  name: aws-auth
  resource_version: "1027"
  self_link: /api/v1/namespaces/kube-system/configmaps/aws-auth
  uid: 0000000-0000-0000-0000-000000000000

$
```

```
$ kubectl get -n kube-system configmap/aws-auth -o yaml > aws-auth.yaml

$ vim aws-auth.yaml

$ kubectl apply -f aws-auth.yaml
configmap/aws-auth created

$
```

```
$ kubectl get -n kube-system configmap/aws-auth
NAME       DATA   AGE
aws-auth   1      2s

$
```

```
$ kubectl get -n kube-system configmap/aws-auth -o yaml
apiVersion: v1
data:
  mapRoles: |
    - rolearn: arn:aws:iam::000000000000:role/eks-cluster-20190729085338669
      username: system:node:{{EC2PrivateDNSName}}
      groups:
      - system:bootstrappers
      - system:nodes
kind: ConfigMap
metadata:
  creation_timestamp: 2019-07-29T08:54:12Z
  name: aws-auth
  resource_version: "1027"
  self_link: /api/v1/namespaces/kube-system/configmaps/aws-auth
  uid: 0000000-0000-0000-0000-000000000000

$
```

```
$ kubectl get -n kube-system configmap/aws-auth -o yaml > aws-auth.yaml

$ vim aws-auth.yaml

$ kubectl apply -f aws-auth.yaml
configmap/aws-auth created

$
```

```
$ kubectl get -n kube-system configmap/aws-auth
NAME       DATA   AGE
aws-auth   1      2s

$
```

```
$ kubectl get -n kube-system configmap/aws-auth -o yaml
apiVersion: v1
data:
  mapRoles: |
    - rolearn: arn:aws:iam::000000000000:role/eks-cluster-20190729085338669
      username: system:node:{{EC2PrivateDNSName}}
      groups:
      - system:bootstrappers
      - system:nodes
kind: ConfigMap
metadata:
  creation_timestamp: 2019-07-29T08:54:12Z
  name: aws-auth
  resource_version: "1027"
  self_link: /api/v1/namespaces/kube-system/configmaps/aws-auth
  uid: 0000000-0000-0000-0000-000000000000

$
```

```
deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-kubernetes
spec:
  selector:
    matchLabels:
      app: hello-kubernetes
  template:
    metadata:
      labels:
        app: hello-kubernetes
    spec:
      containers:
        - name: app
          image: docker/hello-kubernetes:1.8
          ports:
            - containerPort: 8080
```

Please notice that you can find all the Kubernetes resources in the GitHub repository.

You can submit the definition to the cluster with:

```
$ kubectl apply -f deployment.yaml
```

To see if your application runs correctly, you can connect to it with `kubectl port-forward`.

First, retrieve the name of the Pod:

```
$ kubectl get pods
NAME                                     READY   STATUS   RESTARTS   AGE
hello-kubernetes-7859787b7b-vf2j6   1/1     Running   0          6s
$
```

You can connect to the Pod with:

```
$ kubectl port-forward hello-kubernetes-vf2j6 7777:8080
```

The command:

- Connects to the Pod with name `hello-kubernetes-vf2j6`;
- Forwards all the traffic from port 8080 on the Pod to port 8080 on your computer.

If you visit http://localhost:8080 on your computer, you should be greeted by the application.

Great!

Exposing the application with `kubectl port-forward` is an excellent way to test the app quickly, but it isn't a long term solution.

If you wish to route live traffic to the Pod, you should learn a more permanent solution.

In Kubernetes, you can use a Service of `type: LoadBalancer` to expose your Pods.

You can use the following code:

```
service-loadbalancer.yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-kubernetes
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: hello-kubernetes
```

You can submit the YAML file with:

```
$ kubectl apply -f service-loadbalancer.yaml
```

As soon you submit the command, AWS provisions a Classic Load Balancer and connects it to your Pod.

It might take a while for the load balancer to be provisioned.

Eventually, you should be able to describe the Service and retrieve the load balancer's endpoint:

```
$ kubectl describe service hello-kubernetes
Name:                     hello-kubernetes
Namespace:                default
Labels:                   <none>
Annotations:              kubectl.kubernetes.io/last-applied-configuration:
                          {"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"name":"hello-kubernetes","namespace":"default"},"spec":{"ports":[{"port":80,"targetPort":8080}],"selector":{"app":"hello-kubernetes"},"type":"LoadBalancer"}}
Selector:                 app=hello-kubernetes
Type:                     LoadBalancer
IP Families:              <none>
IP:                       172.20.0.1
LoadBalancer Ingress:     ad8f8f0d0d0e4a7a-1473.us-east-1.elb.amazonaws.com
Port:                     <unset>  80/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  32224/TCP
Endpoints:                127.0.0.1:8080
Session Affinity:         None
External Traffic Policy:  Cluster
$
```

If you visit that URL in your browser, you should see the app live.

Excellent!

There's only an issue, though:

The load balancer that you created earlier serves one service at the time.

Also, it has no option to provide intelligent routing based on paths.

So if you have multiple services that need to be exposed, you will need to create the same amount of load balancers.

Imagine having ten applications that have to be exposed.

If you use a Service in `type: LoadBalancer` for each of them, you might end up with ten Classic Load Balancers.

That wouldn't be a problem if those load balancers weren't so expensive.

How can you get around this issue?

In Kubernetes, there's another resource that is designed to solve that problem: the Ingress.

The Ingress has two parts:

1. The first is the **Ingress manifest** which is the same as Deployment or Service in Kubernetes. This is defined by the `kind` part in the YAML manifest.
2. The second part is the **Ingress controller**. This is the actual part that handles the traffic routing, so they know how to serve the requests and forward the data to the Pods.

In other words, the Ingress controller acts as a reverse proxy that routes the traffic to your Pods.

The Ingress routes the traffic based on paths, domains, headers, etc., which consolidates multiple endpoints in a single resource that runs inside Kubernetes.

With this, you can serve multiple services at the same time from one exposed load balancer.

There're several ingress controllers that you can use:

1. Nginx Ingress
2. Ambassador
3. Traefik
4. And more.

In this part you will use the ALB Ingress Controller — an Ingress controller that integrates nicely with the Application Load Balancer.

Routing traffic into the cluster with the ALB Ingress Controller

The ALB Ingress Controller is a Pod that helps you control the Application Load Balancer from Kubernetes.

Instead of setting up Listeners, TargetGroups or Listener Rules from the ALB, you can install the ALB Ingress controller that acts as a translator between Kubernetes and the actual ALB.

In other words, when you create an Ingress manifest in Kubernetes, the controller converts the request into something that the ALB understands (Listeners, TargetGroups, etc.).

Let's have a look at an example:

```
ingress.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-kubernetes
spec:
  rules:
    - http:
        paths:
          - path: /*
            backend:
              serviceName: hello-kubernetes
              servicePort: 80
```

The following Kubernetes Ingress manifest routes all the traffic from path `/` to the Pods targeted by the hello-kubernetes Service.

As soon as you submit the resource to the cluster with `kubectl apply -f ingress.yaml`, the ALB Ingress controller is notified of the new resource.

And it creates:

- A TargetGroup for each Kubernetes Service
- A Listener Rule for the `/*` path in the Ingress resource. This ensures traffic to a specific path is routed to the correct Kubernetes Service.

The ALB Ingress controller is convenient since you can control your infrastructure uniquely from Kubernetes — there's no need to fiddle with AWS anymore.

Now that you know the theory, it's time to put into practice.

First, you should install the ALB Ingress controller.

There're two crucial steps that you need to complete to install the controller:

1. Grant the relevant permissions to your worker nodes
2. Deploy the Kubernetes resources such as Deployment, Service, ConfigMap, etc. necessary to install the controller.

Let's tackle the permissions first:

Since the Ingress controller runs as Pod in one of your Nodes, all the Nodes should have permissions to describe, modify, etc. the Application Load Balancers.

The following file contains the AWS Policy for your workers nodes.

Download the policy and save it in the same folder as your Terraform file `main.tf`.

> You can always refer to the full code in the GitHub repository.

Edit your `main.tf` code and append the following line in your module `module.eks`:

```
main.tf
  source "terr" {
    version = "0.0.1"

    write_kubeconfig  = true
    config_output_path = "./"

    workers_additional_policies = [aws_iam_policy.worker_policy.arn]
  }

  resource "aws_iam_policy" "worker_policy" {
    name   = "worker-policy"
    description = "Worker policy for the ALB Ingress"

    policy = file("iam-policy.json")
  }
```

Before applying the change to the infrastructure, let's do a dry-run with:

```
$ terraform plan
Refreshing state...
Plan: 3 to add, 1 to change, 1 to destroy.
$
```

If you're confident that the change is correct, you can apply with:

```
$ terraform apply
Apply complete! Resources: 1 added, 1 changed, 1 destroyed.
$
```

Great, the first step is completed.

The actual ALB Ingress Controller (file Kubernetes resources such as Pod, ConfigMap, etc.) can be installed with Helm — the Kubernetes package manager.

You should download and install the Helm binary.

You can find the instructions on the official website.

You can verify that Helm was installed correctly with:

```
$ helm version
```

The output should contain the version number.

Helm is a tool that templates and deploys YAML in your cluster.

You can write the YAML yourself, or you can download a package written by someone else.

In this case, you need to install the collection of YAML files necessary to run the ALB Ingress Controller.

First, add the following repository to Helm:

```
$ helm repo add incubator http://storage.googleapis.com/kubernetes-charts-incubator
"incubator" has been added to your repositories
```

Now you can download and install the ALB Ingress Controller in your cluster with:

```
$ helm install incubator/aws-alb-ingress-controller \
  --set autoDiscoverAwsRegion=true
  --set autoDiscoverAwsVpcID=true \
  --set clusterName=MyTestCluster
```

Verify that the Ingress controller is running with:

```
$ kubectl get pods -l "app.kubernetes.io/instance=aggregator"
```

Excellent, you completed step 2 of the installation.

Now you're ready to use the Ingress manifest to route traffic to your app.

You can use the following Ingress manifest definition:

```
ingress.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-kubernetes
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/ssl-redirect: '443'
spec:
  rules:
    - http:
        paths:
          - path: /*
            backend:
              serviceName: hello-kubernetes
              servicePort: 80
```

Pay attention to the following fields:

- `metadata.annotations.kubernetes.io/ingress.class` is used to select the right Ingress controller in the cluster.
- `metadata.annotations.kubernetes.io/ssl.ingress.kubernetes.io/scheme` can be configured to use internal or public-facing load balancers.

> You can explore the full list of annotations here.

You can submit the Ingress manifest to your cluster with:

```
$ kubectl apply -f ingress.yaml
```

It may take a few minutes (the first time) for the ALB to be created, but eventually, you will see the following output from `kubectl describe ingress`:

```
$ kubectl describe ingress hello-kubernetes
Name:             hello-kubernetes
Namespace:        default
Address:          ad8f8f0d0d0e4a7a-1473.us-east-1.elb.amazonaws.com
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host        Path  Backends
  ----        ----  --------
  *           /*    hello-kubernetes:80 (127.0.0.8:8080)
$
```

Excellent, you can use the "Address" to visit your application in the browser.

Congratulations!

You've managed to deploy a fully working cluster that can route live traffic!

Are you done now?

In the spirit of automating all aspects of creating a cluster, is there a way to start the full cluster with a single command?

At the moment, you have to:

1. Provision the cluster with Terraform.
2. Install the ALB Ingress Controller with Helm.

Can you combine those steps?

Provisioning a full cluster with Ingress with the Helm provider

You can extend Terraform with plugins called providers.

So the server-oriented two providers:

1. The AWS provider, to create, modify and delete AWS resources.
2. The Kubernetes provider, as a dependency of the EKS Terraform module.

Terraform has several plugins and one of those is the Helm provider.

What if you could execute Helm from Terraform?

You could create the entire cluster with a single command!

Before you use Helm with Terraform, let's delete the existing Ingress controller with:

```
$ helm delete ingress
```

Let's include Helm in your `main.tf` file like this:

```
main.tf
# ** for bootstrap of main.tf deployment
provider "helm" {
  version = "~> 1.0.0"
  kubernetes {
    host     = data.aws_eks_cluster.cluster.endpoint
    cluster_ca_certificate = base64decode(data.aws_eks_cluster.cluster.cer
    token    = data.aws_eks_cluster_auth.cluster.token
    load_config_file       = false
  }
}

resource "helm_release" "ingress" {
  name  = "ingress"
  chart = "stable/aws-ingress-controller"
  repository = "https://kubernetes-charts.storage.googleapis.com/kubernetes-charts-incubator"
  version    = "0.8.2"

  set {
    name  = "autoscaleConfMaxPeer"
    value = "free"
  }

  set {
    name  = "autoscaleConfMaxCpU"
    value = "free"
  }

  set {
    name  = "clusterName"
    value = local.cluster_name
  }
}
```

> You can find the full code on GitHub.

Terraform has to download and initialise the Helm provider before you can do a `apply`:

```
Term
$ terraform init
$ terraform apply
Terraform initialized
Plan: 1 to add, 1 to change, 1 to destroy.
$
```

You can finally amend your cluster and install the ALB Ingress Controller with a single command:

```
Term
$ terraform apply
$ aws-iam-authenticator -i eks-demo
$ kubectl -n kube-system get deployment
$
```

Excellent, you should verify that the application still works as expected by visiting your app.

If you forgot what the URL was, you could retrieve it with:

```
Term
$ kubectl describe ingress hello-kubernetes
Name:             hello-kubernetes
Namespace:        default
Address:          a8d6d7c3-default-hellokube-d8f0 ap-south-1.elb.amazonaws
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host  Path  Backends
  ----  ----  --------
  *     /     hello-kubernetes:80 (172.16.3.81:8080)
$
```

Success!

If you now destroy your cluster, you can recreate with a single command: `terraform apply`.

There's no need to install and run Helm anymore.

And there's another benefit to having the cluster defined with code and created with a single command.

You can template the Terraform code and create copies of your cluster.

In the next part, you will create three identical environments: dev, staging and production.

Fully automated Dev, Staging, Production environments with Terraform modules

One of the most common tasks when provisioning infrastructure is to create separate environments.

It's a popular practice to provision three environments:

1. A development environment where you can test your changes and integrate them with other colleagues.
2. A staging environment used to sign-off requirements.
3. A production environment.

Since you want your apps to progress through the environments, you might want to provision not one, but three clusters, once for each environment.

When you don't have infrastructure is code, this means coding or clicking your way through three clusters.

But since your cluster creation definition you can refactor your code to create three or more environments with a single command.

The beauty of Terraform is that you can use the same code to generate several clusters with different names.

You can parametrise the name of your resources and create clusters that are exact copies.

You can reuse the existing Terraform code and provision three clusters simultaneously using Terraform modules and Expressions.

> Before you execute the script, it's a good idea to destroy any cluster that you created previously with `terraform destroy`.

The expression syntax is straightforward.

You define a variable like this:

```
example.var.tf
variable "cluster_name" {
  default = "my-cluster"
}
```

You can append the definition at the top of your `main.tf`.

Later, you can reference the variable in the VPC and EKS modules like this:

```
main.tf
module "vpc" {
  # truncated
  cidr = "${var.cluster_name}-vpc"
}

module "eks" {
  # truncated
  cluster_name = "${var.cluster_name}"
}
```

When you execute the usual `terraform apply` command, you can override the variable with a different name.

For example:

```
Term
$ terraform apply -var="cluster_name=dev"
```

The command will provision a new cluster with the name "dev".

In isolation, expressions are not particularly useful.

Let's have a look at an example.

If you execute the following commands, what do you expect?

Is Terraform creating two clusters or update the dev cluster to a staging cluster?

```
Term
$ terraform apply -var="cluster_name=dev"
$ terraform apply -var="cluster_name=staging"
```

The code updates the dev cluster to a staging cluster.

It's overwriting the same cluster!

But what if you wish to create a copy?

You can use the Terraform modules to your advantage.

Terraform modules use variables and expressions to encapsulate resources.

Move your `main.tf` in a subfolder called `cluster` and create an empty `main.tf`.

```
Term
$ mkdir -p cluster
$ mv main.tf cluster/main.tpl.cluster
$ ln -s cluster/main.tf
$ ln -s cluster/*.tpl
$ ln -sf cluster/*.var
$ terraform init
$
```

From now on you can use the code that you've created as a reusable module.

In the root `main.tf` you can reference to that module with:

```
main.tf
module "dev_cluster" {
  source      = "./cluster"
  cluster_name = "dev"
}
```

And since the module is reusable, you can create more than a single cluster:

```
main.tf
module "dev_cluster" {
  source      = "./cluster"
  cluster_name = "dev"
}

module "staging_cluster" {
  source      = "./cluster"
  cluster_name = "staging"
}

module "production_cluster" {
  source      = "./cluster"
  cluster_name = "production"
}
```

You can find the full code changes in the GitHub repository.

You can preview the changes with:

```
Term
$ terraform plan
Plan: 27 to add, 1 to change, 1 to destroy.
$
```

You can apply the changes and create three clusters that are exact copies with:

```
Term
$ terraform apply
```

All three cluster have the ALB Ingress Controller installed, so they are ready to handle production traffic.

What happens when you update the cluster module?

When you modify a property, all clusters will be updated with the same property.

If you wish to customise the properties on a per environment basis, you should extract the parameters to variables and change them from `var.tf`.

Let's have a look at an example.

You might want to run smaller instances such as `t2.micro` in dev and staging and leave the `m4.large` instance type for production.

You can refactor the code and extract the instance type as a variable:

```
main.tf
module "vpc" {
  # truncated
}

  node_groups = {
    first = {
      desired_capacity = 1
      max_capacity     = 10
      min_capacity     = 1

      instance_type = var.instance_type
    }
  }

variable "instance_type" {
  default = "m4.large"
}
```

Later, you can modify the root `main.tf` file with the instance type:

```
main.tf
module "dev_cluster" {
  source      = "./cluster"
  cluster_name = "dev"
  instance_type = "t2.micro"

module "staging_cluster" {
  source      = "./cluster"
  cluster_name = "staging"
  instance_type = "t2.micro"

module "production_cluster" {
  source      = "./cluster"
  cluster_name = "production"
  instance_type = "m4.large"
```

Excellent!

As you can imagine, you can add more variables in your module and create environments with different configurations.

This marks the end of your journey!

A recap on what you've built so far:

1. So you've created an EKS cluster with Terraform.
2. You integrated the ALB Ingress controller as part of the cluster creation.
3. You parametrised the cluster and created a reusable module.
4. You used the module to provision copies of your cluster (one for each environment: dev, staging and production).
5. You made the module more flexible by allowing small customisations such as changing the instance type.

Well done!

Summary and next steps

Having the infrastructure defined as code makes your job easier.

If you wish to change the version of the cluster, you can do it in a centralised manner and have it applied to all clusters.

The setup described above is only the beginning, if you're provisioning production-grade infrastructure you should look into:

- How to structure your Terraform in global and environment-specific layers.
- Managing Terraform state and how to work with the rest of your team.
- How to use External DNS to link your Route53 to your ALB Ingress controller.
- How to set up Fluentd and OLK with ceph storage.
- How to set up a private Elastic Container Registry (ECR).

And the beauty is that External DNS and Cert Manager are available as charts, so you could integrate them with your Helm provider and have all the cluster updated at the same time.

Don't miss the next article!

Be the first to be notified when a new article or Kubernetes experiment is published.

Your email address: Your email address

Subscribe ➔

We'll never share your email address, and you can opt out at any time.

What is LearnK8s?

In-depth Kubernetes training that is practical and easy to understand.

▸ Online courses ❯

Learn Kubernetes online with hands-on, self-paced courses. No need to leave the comfort of your home.

▸ Instructor-led workshops ❯

Deep dive into containers and Kubernetes with the help of our instructors and become an expert in deploying applications at scale.

▸ Corporate training ❯

Train your team in containers and Kubernetes with a customised learning path — remotely or on-site.

LEARNK8S
The Kubernetes training company

SERVICES
Training
Online courses
Self-paced courses
Consulting

COMPANY
Contact us
Team
Careers
Blog
Newsletter

KEEP IN TOUCH

Copyright © learnk8s 2017-2021. Learnk8s® is a trademark. Terms and Conditions | Privacy Policy. Website by Learnk8s.