WS EKS Fargate Using Terraform (EFS, HPA, Ingress, ALB, IRSA, Kubernetes, Helm, Tutorial)

https://www.youtube.com/watch?v=acNFzmblj6U&t=549s

Intro¶

In this video, we're going to go over the following sections:

- **Create AWS VPC Using Terraform**
- **Create AWS EKS Fargate Using Terraform**
- **Update CoreDNS to run on AWS Fargate**
- **Deploy App to AWS Fargate**
- **Deploy Metrics Server to AWS Fargate**
- **Auto Scale with HPA Based on CPU and Memory**
- **Improve Stability with Pod Disruption Budget**
- **Create IAM OIDC provider Using Terraform**
- **Deploy AWS Load Balancer Controller Using Terraform**
- **Create Simple Ingress**
- **Secure Ingress with SSL/TLS**
- **Create Network Loadbalancer**
- **Integrate Amazon EFS with AWS Fargate**

You can find the timestamps in the video description.

Create AWS VPC Using Terraform¶

First of all, we need to declare aws terraform provider. You may want to update the aws **region**, **cluster name**, and possibly a **eks version**. We're also going to be using a new version of the aws provider, so let's set the constrain here as well. I also include terraform lock file (`.terraform.lock.hcl`), so if you encounter any issues, try to copy that file and rerun the terraform.

terraform/0-provider.tf

```
provider "aws" {
  region = "us-east-1"
}

variable "cluster_name" {
  default = "demo"
}

variable "cluster_version" {
  default = "1.22"
}

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

Next, we need to create the AWS **VPC** itself. Here it's very important to enable **dns support** and **hostnames**, especially if you are planning to use the **EFS** file system in your cluster. Otherwise, the **CSI driver** will fail to resolve the EFS endpoint. Currently, AWS Fargate does not support **EBS** volumes, so EFS is the only option for you if you want to run **stateful** workloads in your Kubernetes cluster.

terraform/1-vpc.tf

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"

  # Must be enabled for EFS
  enable_dns_support   = true
  enable_dns_hostnames = true

  tags = {
    Name = "main"
  }
}
```

Then the **Internet Gateway**. It is used to provide internet access directly from the public subnets and indirectly from private subnets by using a NAT gateway.

terraform/2-igw.tf

```
resource
"aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "igw"
  }
}
```

Now we need to create **four** subnets. Two **private** subnets and two **public** subnets. If you are using a different region, you need to update availability zones. Also, it's very important to tag your subnets with the following labels. **Internal-elb** tag used by EKS to select subnets to create private load balancers and **elb** tag for public load balancers. Also, you need to have a cluster tag with owned or shared value.

terraform/3-subnets.tf

```
resource "aws_subnet"
"private-us-east-1a" {
  vpc_id            = aws_vpc.
main.id
  cidr_block        =
"10.0.0.0/19"
  availability_zone = "us-
east-1a"
```

```
  tags = {

"Name"
= "private-us-east-1a"
    "kubernetes.io/role
/internal-elb"            = "1"
    "kubernetes.io/cluster
/${var.cluster_name}" =
"owned"
  }
}

resource "aws_subnet"
"private-us-east-1b" {
  vpc_id            = aws_vpc.
main.id
  cidr_block        =
"10.0.32.0/19"
  availability_zone = "us-
east-1b"

  tags = {

"Name"
= "private-us-east-1b"
    "kubernetes.io/role
/internal-elb"            = "1"
    "kubernetes.io/cluster
/${var.cluster_name}" =
"owned"
  }
}

resource "aws_subnet" "public-
us-east-1a" {
  vpc_id                  =
aws_vpc.main.id
  cidr_block              =
"10.0.64.0/19"
  availability_zone       =
"us-east-1a"
  map_public_ip_on_launch =
true

  tags = {

"Name"
= "public-us-east-1a"
    "kubernetes.io/role
/elb"                     = "1"
```

```
    "kubernetes.io/cluster
/${var.cluster_name}" =
"owned"
  }
}

resource "aws_subnet" "public-
us-east-1b" {
  vpc_id                  =
aws_vpc.main.id
  cidr_block              =
"10.0.96.0/19"
  availability_zone       =
"us-east-1b"
  map_public_ip_on_launch =
true

  tags = {

"Name"
= "public-us-east-1b"
    "kubernetes.io/role
/elb"                    = "1"
    "kubernetes.io/cluster
/${var.cluster_name}" =
"owned"
  }
}
```

For the **NAT Gateway**, I prefer to allocate an Elastic IP address with terraform as well. We need to explicitly depend on the Internet Gateway here and place this NAT to one of the public subnets with a default route to the Internet Gateway.

terraform/4-nat.tf

```
resource "aws_eip" "nat" {
  vpc = true

  tags = {
    Name = "nat"
  }
}

resource "aws_nat_gateway" "nat" {
  allocation_id = aws_eip.nat.id
  subnet_id     = aws_subnet.public-us-east-1a.id

  tags = {
    Name = "nat"
  }

  depends_on = [aws_internet_gateway.igw]
}
```

The last components that we need to create before we can start provisioning EKS are **route tables**. The first is the **private** route table with the default route to the **NAT Gateway**. The second is a **public** route table with the default route to the **Internet Gateway**. Finally, we need to associate previously created subnets with these route tables. Two private subnets and two public subnets.

terraform/5-routes.tf

```
resource "aws_route_table" "private" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat.id
  }

  tags = {
    Name = "private"
  }
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }

  tags = {
    Name = "public"
  }
}

resource "aws_route_table_association" "private-us-east-1a" {
  subnet_id      = aws_subnet.private-us-east-1a.id
  route_table_id = aws_route_table.private.id
}

resource "aws_route_table_association" "private-us-east-1b" {
  subnet_id      = aws_subnet.private-us-east-1b.id
  route_table_id = aws_route_table.private.id
}

resource "aws_route_table_association" "public-us-east-1a" {
  subnet_id      = aws_subnet.public-us-east-1a.id
  route_table_id = aws_route_table.public.id
}

resource "aws_route_table_association" "public-us-east-1b" {
  subnet_id      = aws_subnet.public-us-east-1b.id
  route_table_id = aws_route_table.public.id
}
```

Now, you can declare the rest of the components, including EKS, but I will do it step by step. Let's go to the terminal and run `terraform init` first. Then `terraform apply` to create VPC and subnets.

Create AWS EKS Fargate Using Terraform¶

The next step is to create an **EKS control plane** without any additional nodes. This control plane can be used to attach self-managed, and aws managed nodes as well as you can create **Fargate profiles**.

First of all, let's create an **IAM role** for EKS. It will use it to make API calls to AWS services, for example, to create managed node pools.

terraform/6-eks.tf

```
resource "aws_iam_role" "eks-cluster" {
  name = "eks-cluster-${var.cluster_name}"

  assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "eks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
POLICY
}
```

Then we need to attach **AmazonEKSClusterPolicy** to this role.

terraform/6-eks.tf

```
resource
"aws_iam_role_policy_attachmen
t" "amazon-eks-cluster-
policy" {
  policy_arn = "arn:aws:iam::
aws:policy
/AmazonEKSClusterPolicy"
  role       = aws_iam_role.
eks-cluster.name
}
```

And, of course, the **EKS control plane** itself. You would need to use our role to create a cluster. Also, if you have a **bastion host** or **VPN** configured that allows you to access private IP addresses within the VPC, I would highly recommend enabling a **private endpoint**.

I have a video on how to deploy **OpenVPN** to AWS if you are interested, including how to resolve **private Route53 hosted zones**. If you still decide to use a **public endpoint**, you can restrict access using **CIDR blocks**. Also, specify two private and two public subnets. AWS Fargate can only use **private subnets** with NAT gateway to deploy your pods. Public subnets can be used for load balancers to expose your application to the internet.

terraform/6-eks.tf

```
resource "aws_eks_cluster" "cluster" {
  name     = var.cluster_name
  version  = var.cluster_version
  role_arn = aws_iam_role.eks-cluster.arn

  vpc_config {

    endpoint_private_access = false
    endpoint_public_access  = true
    public_access_cidrs     = ["0.0.0.0/0"]

    subnet_ids = [
      aws_subnet.private-us-east-1a.id,
      aws_subnet.private-us-east-1b.id,
      aws_subnet.public-us-east-1a.id,
      aws_subnet.public-us-east-1b.id
    ]
  }

  depends_on = [aws_iam_role_policy_attachment.amazon-eks-cluster-policy]
}
```

After you provisioned the EKS with terraform, you would need to update your Kubernetes context to access the cluster with the following command. Just update the **region** and **cluster name** to match yours.

```
aws eks update-kubeconfig --name demo --region us-east-1
```

EKS was built to be used as a regular Kubernetes cluster. It expects a **default node pool** to run system components such as **CoreDNS**. If you run `kubectl get pods -A`, you will see that CoreDNS pods are stuck in a **pending state**. Before we can proceed, we need to resolve this issue.

You can verify that you don't have any **nodes** by using this command.

```
kubectl get nodes
```

Update CoreDNS to Run on AWS Fargate¶

To run **CoreDNS** or any other **application** in AWS Fargate, first, you need to create a **Fargate profile**. It is a setting that allows EKS **automatically** create nodes for your application based on Kubernetes namespace and optionally pod labels.

We need to create a single **IAM role** that can be **shared** between all the Fargate profiles. Similar to EKS, Fargate needs permissions to spin up the nodes and connect them to the EKS control plane.

terraform/7-kube-system-profile.tf

```
resource "aws_iam_role" "eks-fargate-profile" {
  name = "eks-fargate-profile"

  assume_role_policy = jsonencode({
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = {
        Service = "eks-fargate-pods.amazonaws.com"
      }
    }]
    Version = "2012-10-17"
  })
}
```

Then we need to attach AWS managed IAM policy called **AmazonEKSFargatePoExecutionRolePolicy**.

terraform/7-kube-system-profile.tf

```
resource "aws_iam_role_policy_attachment" "eks-fargate-profile" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSFargatePodExecutionRolePolicy"
  role       = aws_iam_role.eks-fargate-profile.name
}
```

For the **AWS Fargate profile**, we need to specify the **EKS cluster**. Then the **name**, I usually match it with the Kubernetes namespace and an **IAM role**. When you select subnets for your profile, make sure that you have appropriate tags with cluster name. Finally, you must specify the Kubernetes **namespace** that you want AWS Fargate to manage. Optionally you can filter by pods **labels**.

terraform/7-kube-system-profile.tf

```
resource "aws_eks_fargate_profile" "kube-system" {
  cluster_name           = aws_eks_cluster.cluster.name
  fargate_profile_name   = "kube-system"
  pod_execution_role_arn = aws_iam_role.eks-fargate-profile.arn

  # These subnets must have the following resource tag:
  # kubernetes.io/cluster/<CLUSTER_NAME>.
  subnet_ids = [
    aws_subnet.private-us-east-1a.id,
    aws_subnet.private-us-east-1b.id
  ]

  selector {
    namespace = "kube-system"
  }
}
```

Let's go back to the terminal, and run terraform apply. It should take a minute or two.

```
terraform apply
```

Now, if you get pods again, you would expect that **CoreDNS** should be scheduled already. But most likely, if the EKS team won't fix it in later releases, those coreds pods will continue to be in a **pending** state.

```
kubectl get pods -n kube-system
```

You can try to describe the pod to get some kind of **error** from the Kubernetes controller. You should get something like: **no nodes available**. If you scroll up, you'll see a reason. These pods come with **compute-type: ec2** annotation that prevents them from being scheduled on fargate nodes. The fix is simple, just **remove** the annotation from the deployment template.

First, I'll show you how to fix this **manually** and then a terraform code.

Let's split the terminal, and in the first window run, kubectl get **events**. It's very helpful when you need to debug Kubernetes issues.

```
kubectl get events -w -n kube-system
```

In the second window, just run get pods.

```
watch -n 1 kubectl get pods -n kube-system
```

Then let's use the **kubectl patch** command to remove this annotation from CoreDNS deployment.

```
kubectl patch deployment coredns \
-n kube-system \
--type json \
-p='[{"op": "remove", "path": "/spec/template/metadata/annotations/eks.
amazonaws.com~1compute-type"}]'
```

When you apply, it will immediately recreate CoreDNS **deployment** without those annotations. In a few seconds, AWS Fargate should **spin up** a couple of nodes to fix coredns. It may take up to 5 minutes.

If you rerun kubectl get nodes, you should see two fargate instances. AWS Fargate creates a dedicated node for each pod with similar resource quotas.

```
kubectl get nodes
```

Now, you can patch coredns deployment from the terraform code. But at this time, it will look a little bit ugly. There is a terraform Kubernetes annotation resource, but it only updates high-level annotations, for example, on the deployment object itself. But we need to update **pod-level** annotations. To patch, let's use null resource. It's just a similar kubectl patch command. It works just fine you just need to make sure that you have all the binaries in place, such as **kubectl aws-authenticator**, etc.

terraform/7-kube-system-profile.tf

```
data "aws_eks_cluster_auth" "eks" {
  name = aws_eks_cluster.cluster.id
}

resource "null_resource" "k8s_patcher" {
  depends_on = [aws_eks_fargate_profile.kube-system]

  triggers = {
    endpoint = aws_eks_cluster.cluster.endpoint
    ca_crt   = base64decode(aws_eks_cluster.cluster.certificate_authority[0].data)
    token    = data.aws_eks_cluster_auth.eks.token
  }

  provisioner "local-exec" {
    command = <<EOH
cat >/tmp/ca.crt <<EOF
${base64decode(aws_eks_cluster.cluster.certificate_authority[0].data)}
EOF
kubectl \
  --server="${aws_eks_cluster.cluster.endpoint}" \
  --certificate_authority=/tmp/ca.crt \
  --token="${data.aws_eks_cluster_auth.eks.token}" \
  patch deployment coredns \
  -n kube-system --type json \
  -p='[{"op": "remove", "path": "/spec/template/metadata/annotations/eks.amazonaws.com~1compute-type"}]'
EOH
  }

  lifecycle {
    ignore_changes =
[triggers]
  }
}
```

Since I already fixed this manually, I will **comment this section out**.

Deploy App to AWS Fargate¶

The next step is to deploy an application to AWS Fargate. You already know how to create a **profile**. Let's create another one for **staging** namespace this time.

The only difference here is a **profile name** and a **selector**. With this profile, you can only deploy applications to the **staging namespace**.

terraform/7-kube-system-profile.tf

```
resource
"aws_eks_fargate_profile"
"staging" {
  cluster_name           =
aws_eks_cluster.cluster.name
  fargate_profile_name   =
"staging"
  pod_execution_role_arn =
aws_iam_role.eks-fargate-
profile.arn

  # These subnets must have
the following resource tag:
  # kubernetes.io/cluster
/<CLUSTER_NAME>.
  subnet_ids = [
    aws_subnet.private-us-
east-1a.id,
    aws_subnet.private-us-
east-1b.id
  ]

  selector {
    namespace = "staging"
  }
}
```

Let's quickly apply the terraform.

```
terraform apply
```

Now, let's create another folder for Kubernetes files. The first file is a **simple-deployment.yaml**. In the future, we will use this deployment for **auto-scaling**, and also we will expose it to the internet using the **AWS Load Balancer controller**.

Let's declare a **staging** namespace. Then the deployment object. It's a simple php-apache image that is provided by the Kubernetes community to test autoscaling. The important part here is the **resource block**. AWS Fargate will create a **dedicated node** for your application, so your resource **limit and request should match**.

k8s/simple-deployment.yaml

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: staging
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
  namespace: staging
spec:
  selector:
    matchLabels:
      run: php-apache
  # remove replica if using gitops
  replicas: 1
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: k8s.gcr.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 200m
            memory: 256Mi
          requests:
            cpu: 200m
            memory: 256Mi
```

Alright, let's go and create a deployment. In the first window, run get pods.

```
watch -n 1 -t kubectl get pods -n staging
```

And in the second, use kubectl to apply the deployment.

```
kubectl apply -f k8s/simple-deployment.yaml
```

Deploy Metrics Server to AWS Fargate¶

To be able to **autoscale** our app in Kubernetes, we can use either **Prometheus** or a **metrics server** as a source for CPU and memory usage. Since we're using terraform to provision our infrastructure, let's use terraform **helm provider** to deploy the metrics server as well.

There are multiple ways to **authenticate** with Kubernetes. If you use EKS, the preferred method would be to get a **temporary token** to authenticate with the Kubernetes api server and deploy a **helm chart**. This **aws** command is part of the provider, so you don't need to install anything extra.

terraform/9-metrics-server.tf

```
provider "helm" {
  kubernetes {
    host                   = aws_eks_cluster.cluster.endpoint
    cluster_ca_certificate = base64decode(aws_eks_cluster.cluster.certificate_authority[0].data)
    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      args        = ["eks", "get-token", "--cluster-name", aws_eks_cluster.cluster.id]
      command     = "aws"
    }
  }
}
```

Next is a **helm release**. Give it the name metrics-server. Specify the **namespace**, **version**, and I **disabled internal metrics**, which is actually a default setting. We also need to explicitly depend on the **kube-system** aws fargate profile.

terraform/9-metrics-server.tf

```
resource "helm_release" "metrics-server" {
  name = "metrics-server"

  repository = "https://kubernetes-sigs.github.io/metrics-server/"
  chart      = "metrics-server"
  namespace  = "kube-system"
  version    = "3.8.2"

  set {
    name  = "metrics.enabled"
    value = false
  }

  depends_on = [aws_eks_fargate_profile.kube-system]
}
```

Since we use a **new** helm provider, we need to initialize terraform again and apply after.

```
terraform init
terraform apply
```

If it **fails** to deploy the helm chart, you can check the status of the helm release.

```
helm list -n kube-system
```

Also, check the status of the pod. In case the pod was not able to be scheduled, it may **fail** the terraform.

```
kubectl get pods -n kube-system
```

Auto Scale with HPA Based on CPU and Memory¶

Before we can test auto-scaling, we need to create a service for our deployment. Make sure that you specify the same **namespace** where you deployed the apache.

k8s/service.yaml

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  namespace: staging
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
```

Then fairly simple autoscaling policy. Specify the **minimum** number of pods and the **maximum**. Then the reference to the **deployment object**. For the target, let's just use the **CPU threshold**. If the **average** CPU usage of all pods in this deployment exceeds 50%, the **horizontal pod autoscaller** will add an additional replica.

k8s/hpa.yaml

```yaml
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: staging
spec:
  minReplicas: 1
  maxReplicas: 3
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 50
```

Now, you need to **remove the replica count** from the deployment itself in case you store your config in the **git** and **synchronize with Kubernetes**.

This time we can apply the whole folder.

```
kubectl apply -f k8s/
```

For the demo, again, split the screen. In the first window, we can run get pods. Right now, we have a single replica.

```
watch -n 1 -t kubectl get pods -n staging
```

In the second, you can watch horizontal pod autoscaller. It takes a few seconds for autoscaller to correctly update targets.

```
kubectl get hpa php-apache -w -n staging
```

Lastly, let's run the **load generating tool**. It will spin up an additional pod and continuously run CPU-intensive tasks on apache.

```
kubectl run -i --tty -n staging load-generator --pod-running-timeout=5m0s --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"
```

Now, you can see that the CPU usage goes up. When it **reaches 50%**, we will get a new replica. If it's not enough to bring CPU usage under 50%, autoscaller will create another one. When you use AWS Fargate, you don't need to worry about **cluster autoscaller** since AWS Fargate automatically scales based on the pod requests.

Improve Stability with Pod Disruption Budget¶

Amazon EKS must periodically **patch AWS Fargate pods** to keep them **secure**. Sometimes it means that pods need to be **recreated**. To **limit the impact** on your application, you should set appropriate **pod disruption budgets** (PDBs) to control the number of pods that are down at the same time.

In this example, we limit to **1 unavailable pod** at any given time. You also need to match the label on the deployment object to select appropriate pods. In this case, we use label **run: php-apache**.

k8s/pdb.yaml

```yaml
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: php-apache
  namespace: staging
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      run: php-apache
```

Let's apply it and get **PDBs** in the staging namespace.

```
kubectl apply -f k8s/pdb.yaml
kubectl get pdb -n staging
```

You can see in **real-time** how many pods can be unavailable for this deployment.

Create IAM OIDC provider Using Terraform¶

You can associate an **IAM role** with a **Kubernetes service account**. This service account can then provide **AWS permissions** to the containers in any pod that uses that service account. With this feature, you no longer need to **provide extended permissions** to all Kubernetes nodes so that pods on those nodes can **call AWS APIs**.

terraform/10-iam-oidc.tf

```
data "tls_certificate" "eks" {
  url = aws_eks_cluster.cluster.identity[0].oidc[0].issuer
}

resource "aws_iam_openid_connect_provider" "eks" {
  client_id_list  = ["sts.amazonaws.com"]
  thumbprint_list = [data.tls_certificate.eks.certificates[0].sha1_fingerprint]
  url             = aws_eks_cluster.cluster.identity[0].oidc[0].issuer
}
```

Run terraform init and apply.

```
terraform init
terraform apply
```

You can use **list-open-id-connect-providers** command to find out if the provider was created.

```
aws iam list-open-id-connect-providers
```

Deploy AWS Load Balancer Controller Using Terraform¶

The next step is to deploy **AWS Load Balancer controller**, but first, we need to create an **IAM role** and **establish trust** with the Kubernetes service account.

terraform/11-iam-lb-controller.tf

```
data "aws_iam_policy_document" "aws_load_balancer_controller_assume_role_policy" {
  statement {
    actions = ["sts:AssumeRoleWithWebIdentity"]
    effect  = "Allow"

    condition {
      test     = "StringEquals"
      variable = "${replace(aws_iam_openid_connect_provider.eks.url, "https://", "")}:sub"
      values   = ["system:serviceaccount:kube-system:aws-load-balancer-controller"]
    }

    principals {
      identifiers = [aws_iam_openid_connect_provider.eks.arn]
      type        = "Federated"
    }
  }
}

resource "aws_iam_role" "aws_load_balancer_controller" {
  assume_role_policy = data.aws_iam_policy_document.aws_load_balancer_controller_assume_role_policy.json
  name               = "aws-load-balancer-controller"
}

resource "aws_iam_policy" "aws_load_balancer_controller" {
  policy = file("./AWSLoadBalancerController.json")
  name   = "AWSLoadBalancerController"
}

resource
"aws_iam_role_policy_attachmen
t"
"aws_load_balancer_controller_
attach" {
  role       = aws_iam_role.
aws_load_balancer_controller.
name
  policy_arn = aws_iam_policy.
aws_load_balancer_controller.
arn
}

output
"aws_load_balancer_controller_
role_arn" {
  value = aws_iam_role.
aws_load_balancer_controller.
arn
}
```

Also, let's create **AWSLoadBalancerController** IAM policy. You can get this from the official project for the controller.

terraform/AWSLoadBalancerController.json

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:
CreateServiceLinkedRole"
            ],
            "Resource": "*",
            "Condition": {

"StringEquals": {
                    "iam:
AWSServiceName":
"elasticloadbalancing.
amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:
DescribeAccountAttributes",
                "ec2:
DescribeAddresses",
                "ec2:
DescribeAvailabilityZones",
                "ec2:
DescribeInternetGateways",
                "ec2:
DescribeVpcs",
                "ec2:
DescribeVpcPeeringConnections"
,
                "ec2:
DescribeSubnets",
                "ec2:
DescribeSecurityGroups",
                "ec2:
DescribeInstances",
                "ec2:
DescribeNetworkInterfaces",
                "ec2:
DescribeTags",
                "ec2:
GetCoipPoolUsage",
                "ec2:
DescribeCoipPools",

"elasticloadbalancing:
DescribeLoadBalancers",

"elasticloadbalancing:
DescribeLoadBalancerAttributes
",

"elasticloadbalancing:
DescribeListeners",

"elasticloadbalancing:
DescribeListenerCertificates",

"elasticloadbalancing:
DescribeSSLPolicies",

"elasticloadbalancing:
DescribeRules",

"elasticloadbalancing:
```

```
DescribeTargetGroups",

"elasticloadbalancing:
DescribeTargetGroupAttributes"
,

"elasticloadbalancing:
DescribeTargetHealth",

"elasticloadbalancing:
DescribeTags"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "cognito-idp:
DescribeUserPoolClient",
                "acm:
ListCertificates",
                "acm:
DescribeCertificate",
                "iam:
ListServerCertificates",
                "iam:
GetServerCertificate",
                "waf-regional:
GetWebACL",
                "waf-regional:
GetWebACLForResource",
                "waf-regional:
AssociateWebACL",
                "waf-regional:
DisassociateWebACL",
                "wafv2:
GetWebACL",
                "wafv2:
GetWebACLForResource",
                "wafv2:
AssociateWebACL",
                "wafv2:
DisassociateWebACL",
                "shield:
GetSubscriptionState",
                "shield:
DescribeProtection",
                "shield:
CreateProtection",
                "shield:
```

```
DeleteProtection"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:
AuthorizeSecurityGroupIngress"
,
                "ec2:
RevokeSecurityGroupIngress"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:
CreateSecurityGroup"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:
CreateTags"
            ],
            "Resource": "arn:
aws:ec2:*:*:security-group/*",
            "Condition": {

"StringEquals": {
                    "ec2:
CreateAction":
"CreateSecurityGroup"
                },
                "Null": {
                    "aws:
RequestTag/elbv2.k8s.aws
/cluster": "false"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:
CreateTags",
                "ec2:
DeleteTags"
            ],
            "Resource": "arn:
aws:ec2:*:*:security-group/*",
            "Condition": {
                "Null": {
                    "aws:
RequestTag/elbv2.k8s.aws
/cluster": "true",
                    "aws:
ResourceTag/elbv2.k8s.aws
/cluster": "false"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:
AuthorizeSecurityGroupIngress"
,
                "ec2:
RevokeSecurityGroupIngress",
                "ec2:
DeleteSecurityGroup"
            ],
            "Resource": "*",
            "Condition": {
                "Null": {
                    "aws:
ResourceTag/elbv2.k8s.aws
/cluster": "false"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [

"elasticloadbalancing:
CreateLoadBalancer",

"elasticloadbalancing:
CreateTargetGroup"
            ],
            "Resource": "*",
            "Condition": {
                "Null": {
                    "aws:
```

```
RequestTag/elbv2.k8s.aws
/cluster": "false"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [

"elasticloadbalancing:
CreateListener",

"elasticloadbalancing:
DeleteListener",

"elasticloadbalancing:
CreateRule",

"elasticloadbalancing:
DeleteRule"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [

"elasticloadbalancing:
AddTags",

"elasticloadbalancing:
RemoveTags"
            ],
            "Resource": [
                "arn:aws:
elasticloadbalancing:*:*:
targetgroup/*/*",
                "arn:aws:
elasticloadbalancing:*:*:
loadbalancer/net/*/*",
                "arn:aws:
elasticloadbalancing:*:*:
loadbalancer/app/*/*"
            ],
            "Condition": {
                "Null": {
                    "aws:
RequestTag/elbv2.k8s.aws
/cluster": "true",
                    "aws:
```

```
ResourceTag/elbv2.k8s.aws
/cluster": "false"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [

"elasticloadbalancing:
AddTags",

"elasticloadbalancing:
RemoveTags"
            ],
            "Resource": [
                "arn:aws:
elasticloadbalancing:*:*:
listener/net/*/*/*",
                "arn:aws:
elasticloadbalancing:*:*:
listener/app/*/*/*",
                "arn:aws:
elasticloadbalancing:*:*:
listener-rule/net/*/*/*",
                "arn:aws:
elasticloadbalancing:*:*:
listener-rule/app/*/*/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [

"elasticloadbalancing:
ModifyLoadBalancerAttributes",

"elasticloadbalancing:
SetIpAddressType",

"elasticloadbalancing:
SetSecurityGroups",

"elasticloadbalancing:
SetSubnets",

"elasticloadbalancing:
DeleteLoadBalancer",

"elasticloadbalancing:
```

```
ModifyTargetGroup",

"elasticloadbalancing:
ModifyTargetGroupAttributes",

"elasticloadbalancing:
DeleteTargetGroup"
            ],
            "Resource": "*",
            "Condition": {
                "Null": {
                    "aws:
ResourceTag/elbv2.k8s.aws
/cluster": "false"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [

"elasticloadbalancing:
RegisterTargets",

"elasticloadbalancing:
DeregisterTargets"
            ],
            "Resource": "arn:
aws:elasticloadbalancing:*:*:
targetgroup/*/*"
        },
        {
            "Effect": "Allow",
            "Action": [

"elasticloadbalancing:
SetWebAcl",

"elasticloadbalancing:
ModifyListener",

"elasticloadbalancing:
AddListenerCertificates",

"elasticloadbalancing:
RemoveListenerCertificates",

"elasticloadbalancing:
ModifyRule"
            ],
            "Resource": "*"
        }
    ]
}
```

Finally, we need to deploy controller to Kubernetes using **Helm**. It's going to be deployed to the same kube-system namespace. If you decide to deploy to a different namespace, you also need to create an **AWS Fargate profile** for that **namespace**.

terraform/12-lb-controller.tf

```
resource "helm_release" "aws-
load-balancer-controller" {
  name = "aws-load-balancer-
controller"

  repository = "https://aws.
github.io/eks-charts"
  chart      = "aws-load-
balancer-controller"
  namespace  = "kube-system"
  version    = "1.4.1"

  set {
    name  = "clusterName"
    value = aws_eks_cluster.
cluster.id
  }

  set {
    name  = "image.tag"
    value = "v2.4.2"
  }

  set {
    name  = "replicaCount"
    value = 1
  }

  set {
    name  = "serviceAccount.
name"
    value = "aws-load-
balancer-controller"
  }

  set {
    name  = "serviceAccount.
annotations.eks\\.amazonaws\\.
com/role-arn"
    value = aws_iam_role.
aws_load_balancer_controller.
arn
  }

  # EKS Fargate specific
  set {
    name  = "region"
    value = "us-east-1"
  }

  set {
    name  = "vpcId"
    value = aws_vpc.main.id
  }

  depends_on =
[aws_eks_fargate_profile.kube-
system]
}
```

Now we can apply terraform.

```
terraform apply
```

You can check the helm status.

```
helm list -n kube-system
```

Also, the pod status.

```
kubectl get pods -n kube-system
```

In the following example, we will be using this **ingress class** created by the aws load balancer controller.

```
kubectl get ingressclass
```

Create Simple Ingress¶

Next, let's create ingress to expose apache to the internet using **AWS Load Balancer controller**. First, it's going to be a **plain http**, and in the following example, we will attach a certificate to this load balancer to **terminate TLS**.

By default, **AWS Load Balancer controller** will create load balancers with **private IPs only**. They can only be accessed within your VPC. To change that, we can use **annotations**. In general AWS Load Balancer controller supports two modes. **Instance mode** and **IP mode**. AWS fargate only can be used with IP mode. It will create a target group in AWS and use the pod IP address to route traffic.

k8s/ingress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: php-apache
  namespace: staging
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
spec:
  ingressClassName: alb
  rules:
    - host: php-apache.devopsbyexample.io
      http:
        paths:
          - path: /
            pathType: Exact
            backend:
              service:
                name: php-apache
                port:
                  number: 80
```

I highly recommend when you create your first ingress, **check the logs** on the controller to make sure that there are no errors.

```
kubectl logs -f -n kube-system \
-l app.kubernetes.io/name=aws-load-balancer-controller
```

When you apply the **ingress**, the controller will reconcile and create an **application load balancer**. If you see **permission denied** or a similar **error**, check if the IAM role is properly configured to work with the Kubernetes service account.

```
kubectl apply -f k8s/ingress.yaml
```

To verify the ingress, we need to create a **CNAME record** in your **DNS hosting provider** and point to the load balancer hostname. Check if you can correctly resolve the DNS.

```
dig +short php-apache.devopsbyexample.io
```

Then you can use **curl** or go to the **browser** to access the apache server.

```
curl http://php-apache.devopsbyexample.io
```

Secure Ingress with SSL/TLS¶

It's very unlikely that you would want to expose your service using the plain HTTP protocol. Let's **secure** our apache server with **TLS**.

First, we need to request a certificate from the **AWS Certificate Manager**.

Then we can use either the **autodiscovery mechanism** or explicitly specify the **ARN of the certificate**. You can copy it from the AWS console.

k8s/ingress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: php-apache
  namespace: staging
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-1:<your-acc>:certificate/6b2831b8-3fcc-4b4b-81e8-e7325dfbca84
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80}, {"HTTPS":443}]'
    alb.ingress.kubernetes.io/ssl-redirect: '443'
spec:
  ingressClassName: alb
  rules:
    - host: php-apache.devopsbyexample.io
      http:
        paths:
          - path: /
            pathType: Exact
            backend:
              service:
                name: php-apache
                port:
                  number: 80
```

Now we can apply the ingress.

```
kubectl apply -f k8s/ingress.yam
```

You can check the certificate status in the browser.

Create Network Loadbalancer¶

AWS Load Balancer controller can also manage Kubernetes **service** of type **LoadBalancer**. **In-tree Kuberetnes controller** creates a **classic load balancer**, but the AWS Load Balancer controller will create a **network load balancer**.

k8s/lb.yaml

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache-lb
  namespace: staging
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: external
    service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: ip
    service.beta.kubernetes.io/aws-load-balancer-scheme: internet-facing
    # service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "*"
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    run: php-apache
```

When you apply a service, it should create a network load balancer and provide the **external ip address**.

```
kubectl get svc -n staging
```

To access it, you can use the hostname of the load balancer **directly**.

```
curl http://k8s-staging-phpapach-805ade69e1-e454a6398eb22b41.elb.us-east-1.amazonaws.com
```

Integrate Amazon EFS with AWS Fargate¶

Currently, AWS Fargate **doesn't support** PersistentVolume back by **EBS**. Right now, you can use only **EFS**. With EFS, you can use **ReadWrite Many** mode and mount the same volume to **multiple pods**. With EBS, you can mount a volume only to a single pod.

Let's create an EFS file system using terraform. You can tweak settings based on your requirements; I'll keep them default except for **encryption**.

terraform/13-efs.tf

```
resource
"aws_efs_file_system" "eks" {
  creation_token = "eks"

  performance_mode =
"generalPurpose"
  throughput_mode  =
"bursting"
  encrypted        = true

  # lifecycle_policy {
  #   transition_to_ia =
"AFTER_30_DAYS"
  # }

  tags = {
    Name = "eks"
  }
}

resource
"aws_efs_mount_target" "zone-
a" {
  file_system_id  =
aws_efs_file_system.eks.id
  subnet_id       =
aws_subnet.private-us-east-1a.
id
  security_groups =
[aws_eks_cluster.cluster.
vpc_config[0].
cluster_security_group_id]
}

resource
"aws_efs_mount_target" "zone-
b" {
  file_system_id  =
aws_efs_file_system.eks.id
  subnet_id       =
aws_subnet.private-us-east-1b.
id
  security_groups =
[aws_eks_cluster.cluster.
vpc_config[0].
cluster_security_group_id]
}
```

Let's apply the terraform.

```
terraform apply
```

Now, let's create a **storage class** for EFS. Then the persistent volume. EFS automatically grows and shrinks, but persistent volume **requires** us to provide some sort of capacity. It can by anything. Also, you need to specify your **EFS ID** under **volumeHandle**. You can get it from AWS.

k8s/efs.yaml

```yaml
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: efs-sc
provisioner: efs.csi.aws.com
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: efs-sc
  csi:
    driver: efs.csi.aws.com
    volumeHandle: fs-<your-id>
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-claim
  namespace: staging
spec:
  accessModes:
  - ReadWriteMany
  storageClassName: efs-sc
  resources:
    requests:
      storage: 5Gi
---
apiVersion: v1
kind: Pod
```

```
metadata:
  name: app
  namespace: staging
spec:
  containers:
  - name: app1
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true;
do echo $(date -u) >> /data
/out1.txt; sleep 5; done"]
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: efs-claim
```

In the terminal, for the first time, you can **watch staging events** in case you get an **error**.

```
kubectl get events -n staging
```

When you apply, first, it will take a minute or so for aws fargate to spin up the node, and then it will allocate volume from the EFS file system.

```
kubectl apply -f k8s/efs.yaml
```

If you misconfigured something, the pod would be stuck in a **container-creating** state.

```
kubectl get pods -n staging
```