Artigo
Mikhail Khomenko · Nov. 23, 2020 21min de leitura

Open Exchange

Implantando uma Solução InterSystems IRIS no EKS usando GitHub Actions

Imagine que você queira ver o que a tecnologia InterSystems pode oferecer em termos de análise de dados. Você estudou a teoria e agora quer um pouco de prática. Felizmente, a InterSystems oferece um projeto que contém alguns bons exemplos: Samples BI. Comece com o arquivo README, pulando qualquer coisa associada ao Docker, e vá direto para a instalação passo a passo. Inicie uma instância virtual, instale o IRIS lá, siga as instruções para instalar o Samples BI e, a seguir, impressione o chefe com belos gráficos e tabelas. Por enquanto, tudo bem.

Inevitavelmente, porém, você precisará fazer alterações.

Acontece que manter uma máquina virtual sozinha tem algumas desvantagens e é melhor mantê-la com um provedor em nuvem. A Amazon parece sólida e você cria uma conta no AWS (gratuita para iniciar), lê que usar a identidade do usuário root para tarefas diárias é ruim e cria um usuário IAM normal com permissões de administrador.

Clicando um pouco, você cria sua própria rede VPC, sub-redes e uma instância virtual EC2, e também adiciona um grupo de segurança para abrir a porta web IRIS (52773) e a porta ssh (22) para você. Repete a instalação do IRIS e Samples BI. Desta vez, usa o script Bash ou Python, se preferir. Mais uma vez, impressiona o chefe.

Mas o movimento DevOps onipresente leva você a começar a ler sobre infraestrutura como código e você deseja implementá-la. Você escolhe o Terraform, já que ele é bem conhecido de todos e sua abordagem é bastante universal - adequada com pequenos ajustes para vários provedores em nuvem. Você descreve a infraestrutura em linguagem HCL e traduz as etapas de instalação do IRIS e Samples BI para o Ansible. Em seguida, você cria mais um usuário IAM para permitir que o Terraform funcione. Executa tudo. Ganha um bônus no trabalho.

Gradualmente, você chega à conclusão de que, em nossa era de microsserviços, é uma pena não usar o Docker, especialmente porque a InterSystems lhe diz como. Você retorna ao guia de instalação do Samples BI e lê as linhas sobre o Docker, que não parecem complicadas:

```
  $ docker pull intersystemsdc/iris-community:2019.4.0.383.0-zpm
$ docker run --name irisce -d --publish 52773:52773 intersystemsdc/iris-community:2019.4.0.383.0-zpm
$ docker exec -it irisce iris session iris
USER>zpm
zpm: USER>install samples-bi
```

Depois de direcionar seu navegador para ttp://localhost:52773/csp/user/DeepSee.UserPortal.Home.zen?$NAMESPACE=USER, você vai novamente ao chefe e tira um dia de folga por um bom trabalho.

Você então começa a entender que " docker run " é apenas o começo e você precisa usar pelo menos docker-compose. Não é um problema:

```
  $ cat docker-compose.yml
version: "3.7"
services:
  irisce:
  containername: irisce
  image: intersystemsdc/iris-community:2019.4.0.383.0-zpm
```

```
  ports:
  - 52773:52773
$ docker rm -f irisce # We don ' t need the previous container
$ docker-compose up -d
```

Então, você instala o Docker e o docker-compose com o Ansible e, em seguida, apenas executa o contêiner, que fará o download de uma imagem se ainda não estiver presente na máquina. Em seguida, você instala Samples BI.

Você certamente gosta do Docker, porque é uma interface simples e legal para várias coisas do kernel. Você começa a usar o Docker em outro lugar e geralmente inicia mais de um contêiner. E descobre que muitas vezes os contêineres devem se comunicar entre si, o que leva à leitura sobre como gerenciar vários contêineres.

E você chega ao Kubernetes.

Uma opção para mudar rapidamente de docker-compose para Kubernetes é usar o kompose. Pessoalmente, prefiro simplesmente copiar os manifestos do Kubernetes dos manuais e, em seguida, editá-los, mas o kompose faz um bom trabalho ao concluir sua pequena tarefa:

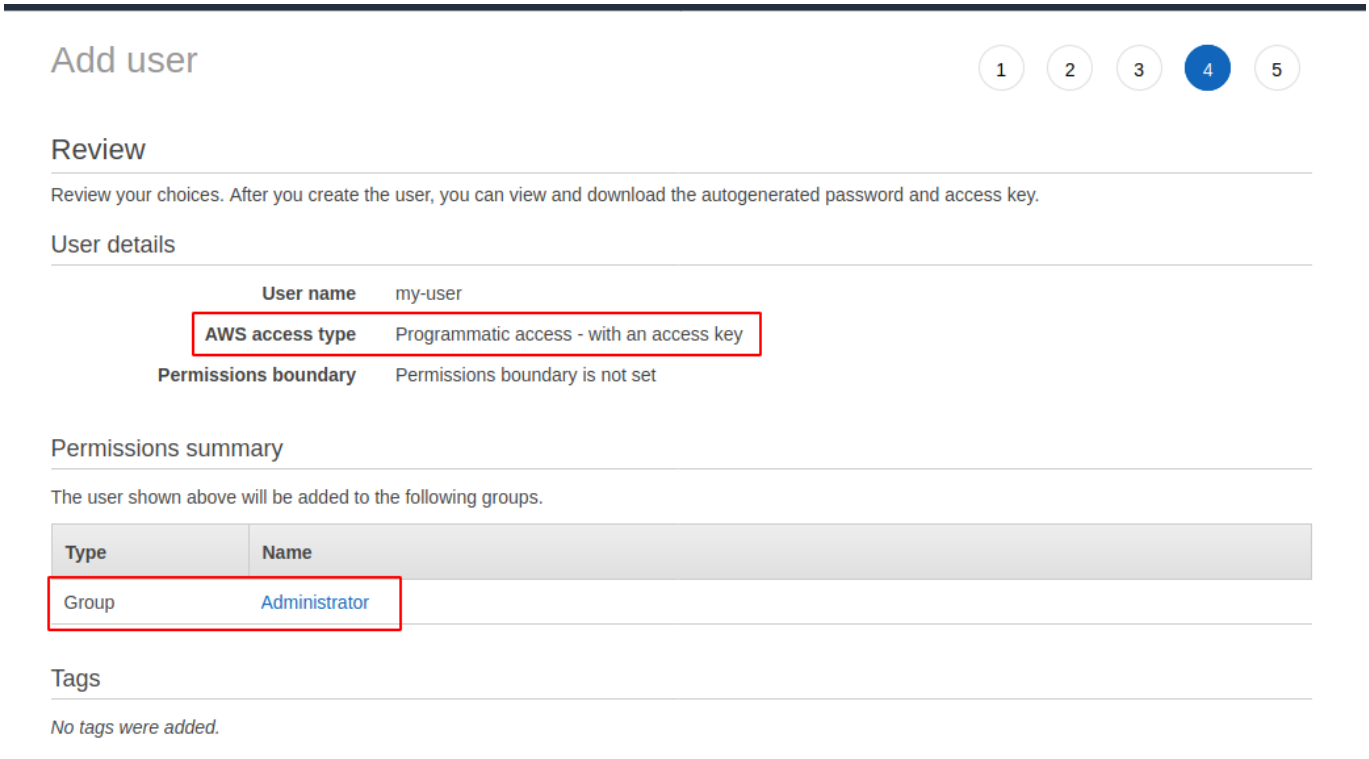
```
  $ kompose convert -f docker-compose.yml
INFO Kubernetes file "irisce-service.yaml" created
INFO Kubernetes file "irisce-deployment.yaml" created
```

Agora você tem os arquivos de implantação e serviço que podem ser enviados para algum cluster do Kubernetes. Você descobre que pode instalar um minikube, que permite executar um cluster Kubernetes de nó único e é exatamente o que você precisa neste estágio. Depois de um ou dois dias brincando com a sandbox do minikube, você está pronto para usar uma implantação real e ao vivo do Kubernetes em algum lugar da nuvem AWS.

Preparação

Então, vamos fazer isso juntos. Neste ponto, faremos algumas suposições:

Primeiro, presumimos que você tenha uma conta no AWS, saiba seu ID e não use credenciais de root. Você criou um usuário IAM (vamos chamá-lo de " my-user ") com direitos de administrador e apenas acesso programático e armazenou suas credenciais. Você também criou outro usuário IAM, chamado " terraform " , com as mesmas permissões:



Em seu nome, o Terraform irá para sua conta no AWS e criará e excluirá os recursos necessários. Os amplos direitos de ambos os usuários são explicados pelo fato de que se trata de uma demonstração. Você salva as credenciais localmente para os dois usuários IAM:

```
  $ cat /.aws/credentials
[terraform]
awsaccesskeyid = ABCDEFGHIJKLMNOPQRST
awssecretaccesskey = ABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890123
[my-user]
awsaccesskeyid = TSRQPONMLKJIHGFEDCBA
awssecretaccesskey = TSRQPONMLKJIHGFEDCBA01234567890123
```

Observação: não copie e cole as credenciais acima. Eles são fornecidos aqui como um exemplo e não existem mais. Edite o arquivo /.aws/credentials e introduza seus próprios registros.

Em segundo lugar, usaremos o ID do conta AWS fictícia (01234567890) para o artigo e a região AWS " eu-west-1. " Sinta-se à vontade para usar outra região.

Terceiro, presumimos que você esteja ciente de que o AWS não é gratuito e que você terá que pagar pelos recursos usados.

Em seguida, você instalou o utilitário AWS CLI para comunicação via linha de comando com o AWS. Você pode tentar usar o aws2, mas precisará definir especificamente o uso do aws2 em seu arquivo de configuração do kube, conforme descrito aqui.

Você também instalou o utilitário kubectl para comunicação via linha de comando com AWS Kubernetes.

E você instalou o utilitário kompose para docker-compose.yml para converter manifestos do Kubernetes.

Finalmente, você criou um repositório GitHub vazio e clonou-o em seu host. Vamos nos referir ao seu diretório raiz como . Neste repositório, vamos criar e preencher três diretórios: .github/workflows/, k8s/, e terraform/.

Observe que todo o código relevante é duplicado no repositório github-eks-samples-bi para simplificar a cópia e a colagem.

Vamos continuar.

Provisionamento AWS EKS

Já conhecemos o EKS no artigo Implementando uma aplicação web simples baseado em IRIS usando o Amazon EKS. Naquela época, criamos um cluster semiautomático. Ou seja, descrevemos o cluster em um arquivo e, em seguida, iniciamos manualmente o utilitário eksctl de uma máquina local, que criou o cluster de acordo com nossa descrição.

O eksctl foi desenvolvido para a criação de clusters EKS e é bom para uma implementação de prova de conceito, mas para o uso diário é melhor usar algo mais universal, como o Terraform. Um ótimo recurso, Introdução ao AWS EKS, explica a configuração do Terraform necessária para criar um cluster EKS. Uma ou duas horas gastas para conhecê-lo não será uma perda de tempo.

Você pode brincar com o Terraform localmente. Para fazer isso, você precisará de um binário (usaremos a versão mais recente para Linux no momento da redação do artigo, 0.12.20) e o usuário IAM " terraform " com direitos suficientes para o Terraform ir para o AWS. Crie o diretório /terraform/ para armazenar o código do Terraform:

```
  $ mkdir /terraform
$ cd /terraform
```

Você pode criar um ou mais arquivos *.tf* (eles são mesclados na inicialização). Basta copiar e colar os exemplos de código da Introdução ao AWS EKS e, em seguida, executar algo como:

```
  $ export AWSPROFILE=terraform
$ export AWSREGION=eu-west-1
$ terraform init
$ terraform plan -out eks.plan
```

Você pode encontrar alguns erros. Nesse caso, brinque um pouco com o modo de depuração, mas lembre-se de desligá-lo mais tarde:

```
  $ export TFLOG=debug
$ terraform plan -out eks.plan

$ unset TFLOG
```

Esta experiência será útil, e muito provavelmente você terá um cluster EKS iniciado (use " terraform apply " para isso). Verifique no console do AWS:

Limpe-o quando você ficar entediado:

```
  $ terraform destroy
```

Em seguida, vá para o próximo nível e comece a usar o módulo Terraform EKS, especialmente porque ele é baseado na mesma introdução ao EKS. No diretório examples/ você verá como usá-lo. Você também encontrará outros exemplos lá.

Simplificamos um pouco os exemplos. Este é o arquivo principal em que os módulos de criação de VPC e de criação de EKS são chamados:

```
  $ cat /terraform/main.tf
terraform {
  requiredversion = ">= 0.12.0"
  backend "s3" {
  bucket = "eks-github-actions-terraform"
  key = "terraform-dev.tfstate"
  region = "eu-west-1"
  dynamodbtable = "eks-github-actions-terraform-lock"
  }
}
provider "kubernetes" {
  host = data.awseksclusterr.cluster.endpoint
  clustercacertificate = base64decode(data.awseksclusterr.cluster.certificateauthority.0.data)
  token = data.awseksclusterauth.cluster.token
  loadconfigfile = false
  version = "1.10.0"
}

locals {
  vpcname = "dev-vpc"
  vpccidr = "10.42.0.0/16"
  privatesubnets = ["10.42.1.0/24", "10.42.2.0/24"]
  publicsubnets = ["10.42.11.0/24", "10.42.12.0/24"]
  clustername = "dev-cluster"
  clusterversion = "1.14"
  workergroupname = "worker-group-1"
  instancetype = "t2.medium"
  asgdesiredcapacity = 1
}

data "awseksclusterr" "cluster" {
  name = module.eks.clusterid
}
```

```
data "awseksclusterauth" "cluster" {
  name = module.eks.clusterid
}

data "awsavailabilityzones" "available" {
}

module "vpc" {
  source = "git::https://github.com/terraform-aws-modules/terraform-aws-vpc?ref=master"

  name = local.vpcname
  cidr = local.vpccidr
  azs = data.awsavailabilityzones.available.names
  privatesubnets = local.privatesubnets
  publicsubnets = local.publicsubnets
  enablenatgateway = true
  singlenatgateway = true
  enablednshostnames = true

  tags = {
  "kubernetes.io/cluster/${local.clustername}" = "shared"
  }

  publicsubnettags = {
  "kubernetes.io/cluster/${local.clustername}" = "shared"
  "kubernetes.io/role/elb" = "1"
  }

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

module "eks" {
  source = "git::https://github.com/terraform-aws-modules/terraform-aws-eks?ref=master"
  clustername = local.clustername
  clusterversion = local.clusterversion
  vpcid = module.vpc.vpcid
  subnets = module.vpc.privatesubnets
  writekubeconfig = false

  workergroups = [
  {
  name = local.workergroupname
  instancetype = local.instancetype
  asgdesiredcapacity = local.asgdesiredcapacity
  }
  ]

  mapaccounts = var.mapaccounts
  maproles = var.maproles
  mapusers = var.mapusers
}
```

Vejamos um pouco mais de perto o bloco "*terraform*" em main.tf:

```
  terraform {
```

```
  requiredversion = ">= 0.12.0"
  backend "s3" {
  bucket = "eks-github-actions-terraform"
  key = "terraform-dev.tfstate"
  region = "eu-west-1"
  dynamodbtable = "eks-github-actions-terraform-lock"
  }
}
```

Aqui, indicamos que seguiremos a sintaxe não inferior ao Terraform 0.12 (muito mudou em comparação com as versões anteriores) e também que Terraform não deve armazenar seu estado localmente, mas sim remotamente, no S3 bucket.

É conveniente se o código do terraform possa ser atualizado de lugares diferentes por pessoas diferentes, o que significa que precisamos ser capazes de bloquear o estado de um usuário, então adicionamos um bloqueio usando uma tabela dynamodb. Leia mais sobre bloqueios na página State Locking (bloqueio de estado).

Como o nome do bucket deve ser único em toda o AWS, o nome " eks-github-actions-terraform " não funcionará para você. Pense por conta própria e certifique-se de que ele ainda não foi usado (se sim, você receberá um erro *NoSuchBucket*):

```
  $ aws s3 ls s3://my-bucket
An error occurred (AllAccessDisabled) when calling the ListObjectsV2 operation: All access to this object has been disabled
$ aws s3 ls s3://my-bucket-with-name-that-impossible-to-remember
An error occurred (NoSuchBucket) when calling the ListObjectsV2 operation: The specified bucket does not exist
```

Tendo criado um nome, crie o bucket (usamos o usuário IAM " terraform " aqui. Ele tem direitos de administrador para que possa criar um bucket) e habilite o controle de versão para ele (o que lhe salvará em caso de um erro de configuração):

```
  $ aws s3 mb s3://eks-github-actions-terraform --region eu-west-1
makebucket: eks-github-actions-terraform
$ aws s3api put-bucket-versioning --bucket eks-github-actions-terraform --versioning-configuration Status=Enabled
$ aws s3api get-bucket-versioning --bucket eks-github-actions-terraform
{
  "Status": "Enabled"
}
```

Com o DynamoDB, ser único não é necessário, mas você precisa criar uma tabela primeiro:

```
  $ aws dynamodb create-table \
  --region eu-west-1 \
  --table-name eks-github-actions-terraform-lock \
  --attribute-definitions AttributeName=LockID,AttributeType=S \
  --key-schema AttributeName=LockID,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

Lembre-se de que, em caso de falha do Terraform, você pode precisar remover um bloqueio manualmente do console AWS. Mas tenha cuidado ao fazer isso.

Com relação aos blocos do módulo eks/vpc em main.tf, a forma de referenciar o módulo disponível no GitHub é simples:

```
  git::https://github.com/terraform-aws-modules/terraform-aws-vpc?ref=master
```

Agora vamos dar uma olhada em nossos outros dois arquivos Terraform (variables.tf e outputs.tf). O primeiro contém nossas variáveis Terraform:

```
  $ cat /terraform/variables.tf
variable "region" {
  default = "eu-west-1"
}
variable "mapaccounts" {
```

```
  description = "Additional AWS account numbers to add to the aws-auth configmap. See
examples/basic/variables.tf for example format."
  type = list(string)
  default = []
}

variable "maproles" {
  description = "Additional IAM roles to add to the aws-auth configmap."
  type = list(object({
  rolearn = string
  username = string
  groups = list(string)
  }))
  default = []
}

variable "mapusers" {
  description = "Additional IAM users to add to the aws-auth configmap."
  type = list(object({
  userarn = string
  username = string
  groups = list(string)
  }))
  default = [
  {
  userarn = "arn:aws:iam::01234567890:user/my-user"
  username = "my-user"
  groups = ["system:masters"]
  }
  ]
}
```

A parte mais importante aqui é adicionar o usuário IAM “ my-user ” à variável mapusers, mas você deve usar seu próprio ID de conta aqui no lugar de 01234567890.

O que isto faz? Quando você se comunica com o EKS por meio do cliente kubectl local, ele envia solicitações ao servidor da API Kubernetes, e cada solicitação passa por processos de autenticação e autorização para que o Kubernetes possa entender quem enviou a solicitação e o que elas podem fazer. Portanto, a versão EKS do Kubernetes pede ajuda ao AWS IAM com a autenticação do usuário. Se o usuário que enviou a solicitação estiver listado no AWS IAM (apontamos seu ARN aqui), a solicitação vai para a fase de autorização, que o EKS processa sozinho, mas de acordo com nossas configurações. Aqui, indicamos que o usuário IAM “ my-user ” é muito legal (grupo “ system: masters).”

Por fim, o arquivo outputs.tf descreve o que o Terraform deve imprimir após concluir um job:

```
  $ cat /terraform/outputs.tf
output "clusterendpoint" {
  description = "Endpoint for EKS control plane."
  value = module.eks.clusterendpoint
}
output "clustersecuritygroupid" {
  description = "Security group ids attached to the cluster control plane."
  value = module.eks.clustersecuritygroupid
}

output "configmapawsauth" {
  description = "A kubernetes configuration to authenticate to this EKS cluster."
  value = module.eks.configmapawsauth
}
```

Isso completa a descrição da parte do Terraform. Voltaremos em breve para ver como vamos iniciar esses arquivos.

Manifestos Kubernetes

Até agora, cuidamos de *onde* iniciar a aplicação. Agora vamos ver *o que* executar.

Lembre-se de que temos o docker-compose.yml (renomeamos o serviço e adicionamos alguns rótulos que o kompose usará em breve) no diretório /k8s/:

```
  $ cat /k8s/docker-compose.yml
version: "3.7"
services:
  samples-bi:
  containername: samples-bi
  image: intersystemsdc/iris-community:2019.4.0.383.0-zpm
  ports:
  - 52773:52773
  labels:
  kompose.service.type: loadbalancer
  kompose.image-pull-policy: IfNotPresent
```

Execute o kompose e adicione o que está destacado abaixo. Exclua as anotações (para tornar as coisas mais inteligíveis):

```
  $ kompose convert -f docker-compose.yml --replicas=1
$ cat /k8s/samples-bi-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
  io.kompose.service: samples-bi
  name: samples-bi
spec:
  replicas: 1
  strategy:
  type: Recreate
  template:
  metadata:
  labels:
  io.kompose.service: samples-bi
  spec:
  containers:
  - image: intersystemsdc/iris-community:2019.4.0.383.0-zpm
  imagePullPolicy: IfNotPresent
  name: samples-bi
  ports:
  - containerPort: 52773
  resources: {}
  lifecycle:
  postStart:
  exec:
  command:
  - /bin/bash
  - -c
  - |
  echo -e "write \nhalt" > test
  until iris session iris echo -e "zpm \ninstall samples-bi \nquit \nhalt" > samplesbiinstall
  iris session iris rm test samplesbiinstall
```

```
  restartPolicy: Always
```

Usamos a estratégia de atualização de Recriar, o que significa que o pod será excluído primeiro e depois recriado. Isso é permitido para fins de demonstração e nos permite usar menos recursos.
Também adicionamos o postStart hook, que será disparado imediatamente após o início do pod. Esperamos até o IRIS iniciar e instalar o pacote samples-bi do repositório zpm padrão.
Agora adicionamos o serviço Kubernetes (também sem anotações):

```
  $ cat /k8s/samples-bi-service.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
  io.kompose.service: samples-bi
  name: samples-bi
spec:
  ports:
  - name: "52773"
  port: 52773
  targetPort: 52773
  selector:
  io.kompose.service: samples-bi
  type: LoadBalancer
```

Sim, vamos implantar no namespace "padrão", que funcionará para a demonstração.

Ok, agora sabemos *onde* e *o que* queremos executar. Resta ver *como*.

O fluxo de trabalho do GitHub Actions

Em vez de fazer tudo do zero, criaremos um fluxo de trabalho semelhante ao descrito em Implantando uma solução InterSystems IRIS no GKE usando GitHub Actions. Desta vez, não precisamos nos preocupar em construir um contêiner. As partes específicas do GKE são substituídas pelas específicas do EKS. As partes em negrito estão relacionadas ao recebimento da mensagem de commit e ao uso dela em etapas condicionais:

```
  $ cat /.github/workflows/workflow.yaml
name: Provision EKS cluster and deploy Samples BI there
on:
  push:
  branches:
  - master
# Environment variables.
# ${{ secrets }} are taken from GitHub -> Settings -> Secrets
# ${{ github.sha }} is the commit hash
env:
  AWSACCESSKEYID: ${{ secrets.AWSACCESSKEYID }}
  AWSSECRETACCESSKEY: ${{ secrets.AWSSECRETACCESSKEY }}
  AWSREGION: ${{ secrets.AWSREGION }}
  CLUSTERNAME: dev-cluster
  DEPLOYMENTNAME: samples-bi

jobs:
  eks-provisioner:
  # Inspired by:
  ## https://www.terraform.io/docs/github-actions/getting-started.html
  ## https://github.com/hashicorp/terraform-github-actions
  name: Provision EKS cluster
  runs-on: ubuntu-18.04
  steps:
  - name: Checkout
  uses: actions/checkout@v2
```

```
- name: Get commit message
run: |
echo ::set-env name=commitmsg::$(git log --format=%B -n 1 ${{ github.event.after }})

- name: Show commit message
run: echo $commitmsg

- name: Terraform init
uses: hashicorp/terraform-github-actions@master
with:
tfactionsversion: 0.12.20
tfactionssubcommand: 'init'
tfactionsworkingdir: 'terraform'

- name: Terraform validate
uses: hashicorp/terraform-github-actions@master
with:
tfactionsversion: 0.12.20
tfactionssubcommand: 'validate'
tfactionsworkingdir: 'terraform'

- name: Terraform plan
 if: "!contains(env.commitmsg, '[destroy eks]')"
uses: hashicorp/terraform-github-actions@master
with:
tfactionsversion: 0.12.20
tfactionssubcommand: 'plan'
tfactionsworkingdir: 'terraform'

- name: Terraform plan for destroy
if: "contains(env.commitmsg, '[destroy eks]')"
uses: hashicorp/terraform-github-actions@master
with:
tfactionsversion: 0.12.20
tfactionssubcommand: 'plan'
args: '-destroy -out=./destroy-plan'
tfactionsworkingdir: 'terraform'

- name: Terraform apply
if: "!contains(env.commitmsg, '[destroy eks]')"
uses: hashicorp/terraform-github-actions@master
with:
tfactionsversion: 0.12.20
tfactionssubcommand: 'apply'
tfactionsworkingdir: 'terraform'

- name: Terraform apply for destroy
 if: "contains(env.commitmsg, '[destroy eks]')"
uses: hashicorp/terraform-github-actions@master
with:
tfactionsversion: 0.12.20
tfactionssubcommand: 'apply'
args: './destroy-plan'
tfactionsworkingdir: 'terraform'

kubernetes-deploy:
name: Deploy Kubernetes manifests to EKS
needs:
```

```
  - eks-provisioner
  runs-on: ubuntu-18.04
  steps:
  - name: Checkout
  uses: actions/checkout@v2

  - name: Get commit message
  run: |
  echo ::set-env name=commitmsg::$(git log --format=%B -n 1 ${{ github.event.after }})

  - name: Show commit message
  run: echo $commitmsg

  - name: Configure AWS Credentials
  if: "!contains(env.commitmsg, '[destroy eks]')"
  uses: aws-actions/configure-aws-credentials@v1
  with:
  aws-access-key-id: ${{ secrets.AWSACCESSKEYID }}
  aws-secret-access-key: ${{ secrets.AWSSECRETACCESSKEY }}
  aws-region: ${{ secrets.AWSREGION }}

  - name: Apply Kubernetes manifests
  if: "!contains(env.commitmsg, '[destroy eks]')"
  working-directory: ./k8s/
  run: |
  aws eks update-kubeconfig --name ${CLUSTERNAME}
  kubectl apply -f samples-bi-service.yaml
  kubectl apply -f samples-bi-deployment.yaml
  kubectl rollout status deployment/${DEPLOYMENTNAME}
```

É claro que, precisamos definir as credenciais do usuário "terraform" (retirá-las do arquivo ~/.aws/credentials), permitindo que o GitHub use seus segredos:

Observe as partes destacadas do fluxo de trabalho. Elas nos permitirão destruir um cluster EKS empurrando uma mensagem de commit que contém a frase " [destroy eks] " . Observe que não executaremos "kubernetes apply" com essa mensagem de commit.
Execute um pipeline, mas primeiro crie um arquivo .gitignore:

```
  $ cat /.gitignore
.DSStore
terraform/.terraform/
terraform/*.plan
terraform/*.json
$ cd
$ git add .github/ k8s/ terraform/ .gitignore
$ git commit -m "GitHub on EKS"
$ git push
```

Monitore o processo de implantação na aba "Actions" da página do repositório GitHub. Aguarde a conclusão com sucesso.

Quando você executa um fluxo de trabalho pela primeira vez, leva cerca de 15 minutos na etapa " Terraform apply " , aproximadamente o mesmo tempo necessário para criar o cluster. Na próxima inicialização (se você não excluiu o cluster), o fluxo de trabalho será muito mais rápido. Você pode verificar isso:

```
  $ cd
$ git commit -m "Trigger" --allow-empty
$ git push
```

É claro que seria bom verificar o que fizemos. Desta vez, você pode usar as credenciais do IAM " my-user " em seu

computador:
$ export AWSPROFILE=my-user
$ export AWSREGION=eu-west-1
$ aws sts get-caller-identity
$ aws eks update-kubeconfig --region=eu-west-1 --name=dev-cluster --alias=dev-cluster
$ kubectl config current-context
dev-cluster
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
ip-10-42-1-125.eu-west-1.compute.internal Ready 6m20s v1.14.8-eks-b8860f

$ kubectl get po
NAME READY STATUS RESTARTS AGE
samples-bi-756dddffdb-zd9nw 1/1 Running 0 6m16s

$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 172.20.0.1 443/TCP 11m
samples-bi LoadBalancer 172.20.33.235 **a2c6f6733557511eab3c302618b2fae2-622862917.eu-west-1.elb.amazonaws.com** 52773:31047/TCP 6m33s

Vá para http://a2c6f6733557511eab3c302618b2fae2-622862917.eu-west-1.elb.amazonaws.com:52773/csp/user/DeepSee.UserPortal.Home.zen?$NAMESPACE=USER (substitua o link pelo seu IP externo), então, digite “ system ”, “ SYS ” e altere a senha padrão. Você deve ver vários painéis de Inteligência Empresarial (BI):

Clique na seta de cada um para um mergulho mais profundo:

Lembre-se, se você reiniciar um pod samples-bi, todas as suas alterações serão perdidas. Este é um comportamento intencional, pois esta é uma demonstração. Se você precisar de persistência, criei um exemplo no repositório github-gke-zpm-registry/k8s/statefulset.tpl.

Quando terminar, basta remover tudo que você criou:
$ git commit -m "Mr Proper [destroy eks]" --allow-empty
$ git push

Conclusão

Neste artigo, substituímos o utilitário eksctl pelo Terraform para criar um cluster EKS. É um passo à frente para “ codificar ” toda a sua infraestrutura AWS.
Mostramos como você pode facilmente implantar uma aplicação de demonstração com git push usando GitHub Actions e Terraform.
Também adicionamos o kompose e um postStart hook de um pod à nossa caixa de ferramentas.
Não mostramos a ativação do TLS neste momento. Essa é uma tarefa que realizaremos em um futuro próximo.

#AWS #Conteinerização #DevOps #Docker #Kubernetes #Nuvem #InterSystems IRIS #Open Exchange
Confira o aplicativo relacionado no InterSystems Open Exchange

URL de origem:https://pt.community.intersystems.com/post/implantando-uma-solu%C3%A7%C3%A3o-intersystems-iris-no-eks-usando-github-actions