記事
Toshihiko Minamoto · 2021年1月6日 24m read

Open Exchange

GitHub Actions を使って EKS に InterSystems IRIS Solution をデプロイする

データ分析のためにInterSystemsで何ができるかを確認したいとしましょう。 理論を学んだので今度は実践したいと考えた時、InterSystems には、役に立つ実例をいくつか取り入れた Samples BI というプロジェクトが用意されています。 まずは、README ファイルを開き、Docker に関する内容はすべてスキップして、手順に従ってインストールしてゆきます。
仮想インスタンスを起動し、そこに IRIS をインストールしたら、手順に従って Samples BI をインストールします。そして綺麗なチャートやテーブルを見せて上司に感心してもらいましょう。
ここまでは順調ですね。

必然的に、ここで変更を加えることになります。

仮想マシンを自分でメンテすることにはデメリットがいくつかあり、クラウドプロバイダーに保管する方が無難であることが判明します。 Amazon は安定してそうなので、AWS アカウントを作成します (開始は無料)。
ルートユーザーの ID を使って日々の業務を行うのは危険であることを読んだあなたは、管理者権限を持つ普通の IAM ユーザーを作成します。

少し先に進んでから、独自の VPC ネットワーク、サブネット、EC2 の仮想インスタンスを作成し、さらには自分用の IRIS ウェブポート (52773) と SSH ポート (22) を開くために、セキュリティグループを追加します。 IRIS と Samples BI のインストールをもう一度実行します。 今回は、Bash シェルスクリプトを使います。Python でも OK。
また、ここでも上司に好印象を与えておきます。

しかし、DevOps がトレンドとなり、Infrastructure as Code (コードによるインフラの管理) について学び、導入したくなります。 そこで、Terraform を選択します。知名度が高く、世界中で多用されている上に、多くのクラウドプロバイダーにとっても調整が簡単なため適切な選択肢です。 インフラストラクチャを HCL 言語で定義し、IRIS と Samples BI のインストール手順を Ansible に変換します。 そしてTerraform を動かすIAM ユーザーをもう 1 つ作成し、それを実行します。そして 職場でボーナスをゲットします。

次第に、マイクロサービスの時代に Docker を使わないのはもったいない、InterSystems がそのドキュメンテーションを提供してくれることを考えるとなおさらだ、という結論に至ります。 Samples BI のインストールガイドに戻り、複雑に思えなくなったDocker について書かれた行を読み通します。

```
  $ 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
```

ブラウザーを http://localhost:52773/csp/user/DeepSee.UserPortal.Home.zen?$NAMESPACE=USER, に変更したら、もう一度上司の所に戻り、良い仕事のご褒美として1 日休みをもらいます。

あなたはその時、“ docker run ”は単なる第一歩であり、少なくともdocker-compose を使う必要があるでは？と考え始めます。 問題ありません:

```
  $ 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
```

こうして、Ansible を使ってDocker や docker-compose
をインストールし、マシン上にイメージが無ければダウンロードするコンテナを単に実行します。
それから、Samples BI をインストールします。

カーネルの様々な機能へのインターフェイスとして活躍する、便利でシンプルな Docker
を気に入らないはずがありません。 あなたは Docker
を他の場所でも使い、頻繁に複数のコンテナを起動するようにもなりました。 また、コンテナはしばしばお互いに通信し合う必要があることを知ったあなたは、複数のコンテナを管理する方法について読み始めます。

そして、Kubernetes (クーベネティス) について知ります。

docker-compose から Kubernetes に素早く切り替える方法の 1 つに kompose を使うという方法があります。
私個人的には、Kubernetes
のマニフェストファイルをマニュアルからコピーし、自分で編集する方が好きなのですが、kompose
でも小さなタスクを完了させるだけなら問題ありません。

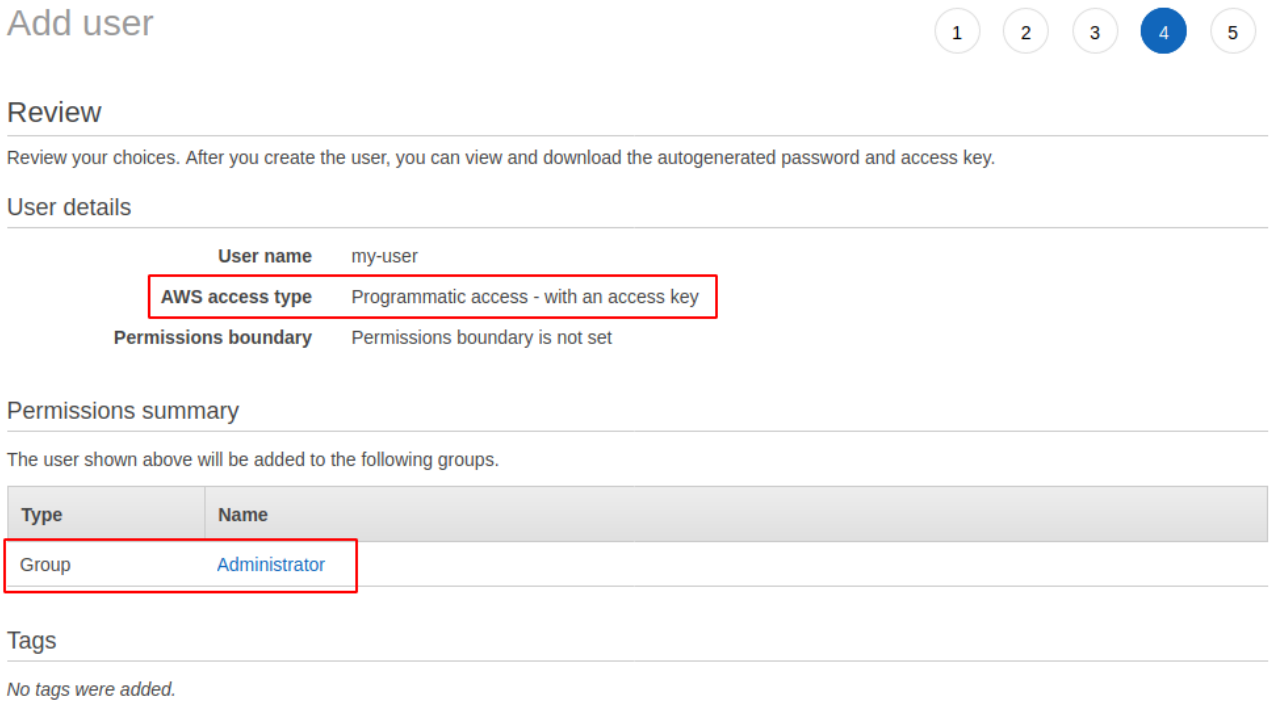
```
  $ kompose convert -f docker-compose.yml
INFO Kubernetes file "irisce-service.yaml" created
INFO Kubernetes file "irisce-deployment.yaml" created
```

これで、Kubernetes のクラスターに送信できるデプロイファイルとサービスファイルができました。
ここであなたは、minikube をインストールすれば、シングルノードの Kubernetes
クラスターを実行できること、そしてこの段階ではまさにそれが必要なことを知ります。 minikube
のサンドボックスを 1 日か 2 日使ってみた後、ライブの Kubernetes を実際に AWS
クラウドのどこかでデプロイする準備が整いました。

セットアップ

それでは、一緒にやりましょう。 この時点で、前提にしておきたい点がいくつかあります。

第一に、あなたが AWS のアカウントを持っていること、その ID を知っている
こと、そしてルートユーザーの認証情報を使用していないということです。 管理者権限
とプログラムによるアクセス権だけを持つ IAM ユーザー (「my-user」とでも呼びましょう)
を作成し、その認証情報を保管します。 また、「terraform」と名付けた IAM
ユーザーも作成し、同じアクセス権を与えます。



このユーザーの代わりに、Terraform はあなたの AWS
アカウントにアクセスし、必要なリソースを作成、削除します。
これはデモなので、これらのユーザーに広範な権限を与えています。 両方の IAM
ユーザーの認証情報はローカルに保存します。
 $ cat ~/.aws/credentials
[terraform]
aws_access_key_id = ABCDEFGHIJKLMNOPQRST
aws_secret_access_key = ABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890123
[my-user]
aws_access_key_id = TSRQPONMLKJIHGFEDCBA
aws_secret_access_key = TSRQPONMLKJIHGFEDCBA01234567890123

注意: 上の認証情報はコピー & ペーストしないでください。
例として作成したものなので、既に削除しています。 ~/.aws/credentials
ファイルを編集し、独自のレコードを導入してください。

第二に、この記事では AWS アカウントのダミー ID に「01234567890」を、AWS の地域には「eu-west-1」を使用します。 別の地域を使っても構いません。

第三に、あなたが AWS は有料
であること、使用するリソースは支払う必要があることを認識しているという前提で進めて行きます。

次に、あなたはコマンドラインを使って AWS と通信するために、 AWS CLI ユーティリティ
をインストールしています。 aws2 を使うこともできますが、kube の config ファイルに aws2
の使用範囲を具体的に設定する必要があります。詳しくは、こちらをご覧ください。

また、コマンドラインを使って AWS Kubernetes と通信する目的で、 kubectl ユーティリティ
もインストールしています。

さらに、Kubernetes のマニフェストファイルを変換するには、docker-compose.yml
を使用する必要があり、そのためには kompose ユーティリティをインストールしなくてはいけません。

最後に、空の GitHub レポジトリを作成し、ホストにクローンしました。 ルートディレクトリは .
として参照します。 このリポジトリでは、「.github/workflows/」、「k8s/」、「terraform/」という 3

つのディレクトリを作成し、中身を埋めていきます。

すべての関連するコードは、コピー & ペーストを簡単にできるよう、github-eks-samples-bi
というリポジトリ内に複製されています。

それでは続けましょう。

AWS EKS のプロビジョニング

EKS については、Amazon EKS を使ったシンプルな IRIS ベースのウェブアプリケーションをデプロイする
と題した記事で既に紹介しています。 そのときに、クラスターを半自動的に作成しました。
つまり、クラスタをファイル内で定義し、 eksctl ユーティリティ
を手動でローカルマシンから起動した結果、クラスタが私たちの定義に従って作成されています。

eksctl は EKS クラスタを作成する目的で開発され、概念実証による導入には優れていますが、日々の使用には
Terraform など、より広範な機能を提供するツールを採用する方が無難と言えます。 AWS EKS Introduction
と題した非常に便利なリソースがあり、EKS クラスタの作成に必要な Terraform の設定が説明されています。
1、2 時間かけて読む価値は十分にあります。

Terraform はローカルで試すことができます。 そのためには、バイナリ (この記事の執筆時には Linux
の最新バージョン 0.12.20 を使用しました) と Terraform を AWS にアクセスさせるに十分な権限を持つ IAM
ユーザー「terraform」が必要です。 ディレクトリ「/terraform/」を作成し、 次の Terraform
のコードを保管します。

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

.tf ファイルは、複数作成できます (起動時にマージされます)。 AWS EKS Introduction
に記載されているコードの例をコピー & ペーストし、以下のようなコードを実行します。

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

何らかのエラーが発生する可能性があります。
その場合は、デバッグモードを試してください。終わったら忘れずにオフにしてください。

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

$ unset TFLOG
```

この体験はお役に立つと思います。また、EKS クラスタも起動できるでしょう (「terraform
apply」を使ってください)。 AWS コンソールでお試しください。

飽きてしまったらクリーンアップしましょう。

```
  $ terraform destroy
```

そして、次のレベルに移動し、Terraform EKS モジュールの使用を開始します。これは、同じ EKS Introduction
がベースになっているためでもあります。 その使い方は、「examples/」ディレクトリで確認してください。
そこには、他の例も用意してあります。

例はある程度簡素化してあります。 こちらが、VPC の作成モジュールと EKS
の作成モジュールが呼び出されるメインファイルです。

```
  $ 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
}
```

それでは、main.tf の「*terraform*」ブロックをもう少し細かく見てみましょう。

```
  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"
  }
}
```

ここでは、Terraform 0.12 以上のバージョン (以前のバージョンに比べると多くの変更が加えられています) の構文に従うこと、また Terraform はその状態をローカルにではなく、むしろリモートの S3 バケットに保管することを指定しています。

Terraform のコードを色々な人が色々な場所から更新できると便利ですが、ユーザーの状態をロックできなければいけないということになるので、dynamodb テーブルを使ってロックを追加しました。 ロックの詳細は、State Locking のページをご覧ください。

バケットには AWS 全体で一意の名前が使われている必要があるので「eks-github-actions-terraform」という名前は使えません。 独自の名前を考えて、既に使用されていないことを確認してください (すると *NoSuchBucket* エラーが発生します):

```
  $ 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
```

名前を考えたら、バケットを作成し (ここでは IAM ユーザー「terraform」を使います。管理者権限を持っているのでバケットを作成できます)、そのバージョン管理を有効にします (こうすることで、構成エラーが出ても落ち着いて対応できます)。

```
  $ 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"
}
```

DynamoDB では、一意性は必要ありませんが、最初にテーブルを作成する必要があります。

```
  $ 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
```

Terraform に障害が発生した場合は、ロックを AWS コンソールから手動で解除する必要があるかもしれません。その場合は慎重に対応してください。

main.tf 内にある eks/vpc モジュールのブロックについてですが、GitHub で使用可能なモジュールを参照する方法はいたってシンプルです。

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

それでは、他の 2 つの Terraform ファイル (variables.tf と outputs.tf) を見てみましょう。 1 つ目のファイルには Terraform の変数が置かれます。

```
  $ cat /terraform/variables.tf
variable "region" {
  default = "eu-west-1"
}
variable "mapaccounts" {
  description = "aws-auth configmap に追加する他の AWS アカウント番号。
フォーマットの例は、examples/basic/variables.tf を参照してください。"
  type = list(string)
  default = []
}

variable "maproles" {
  description = "aws-auth configmap に追加する他の IAM ロール。"
  type = list(object({
  rolearn = string
  username = string
  groups = list(string)
  }))
  default = []
}

variable "mapusers" {
  description = "aws-auth configmap に追加する他の IAM ユーザー。"
  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"]
  }
```

```
 ]
}
```

ここで一番大事なのは mapusers 変数に IAM ユーザー “ my-user ” を追加するということですが、01234567890 の代わりに自分のアカウント ID を使ってください。

これで何が起こるのか？ ローカルの kubectl クライアントを通じて EKS と通信する場合は、EKS から Kubernetes API
サーバーにリクエストが送られます。そして、各リクエストは認証プロセスと承認プロセスを通り、Kubernetes はリクエストの送信者とそのユーザーの権限を理解できるようになります。 そして、Kubernetes の EKS バージョンは、AWS の IAM にユーザー認証のサポートを求めます。 リクエストを送信したユーザーが AWS IAM (ここではユーザーの ARN をポイントしました) に載っている場合、リクエストは承認ステージに移動し、EKS が私たちの設定に従って直接処理します。 ここでは、IAM ユーザー “ my-user ” は信頼できる人である group “ system: masters ”、” ことを示しています。

最後に、Terraform がジョブを完了した後に出力する内容を outputs.tf が説明します。

```
 $ cat /terraform/outputs.tf
output "clusterendpoint" {
 description = "EKS コントロールプレーンのエンドポイント"
 value = module.eks.clusterendpoint
}
output "clustersecuritygroupid" {
 description = "クラスターのコントロールプレーンに関連付けられたセキュリティグループの ID。"
 value = module.eks.clustersecuritygroupid
}

output "configmapawsauth" {
 description = "この EKS クラスターに対して承認する Kubernetes 構成。"
 value = module.eks.configmapawsauth
}
```

これで Terraform の部分に関する説明は終了です。
これらのファイルを起動する方法はもう少し後で説明します。

Kubernetes のマニフェスト

ここまでは、アプリケーションをどこで起動するのかについて説明しましたが、
今度は、*何*を実行するのか、を見て行きます。

docker-compose.yml (サービス名を変更し、kompose が使用するラベルをいくつか追加しました) が、 /k8s/ ディレクトリにあることを覚えているでしょうか。

```
 $ 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
```

kompose を実行して、下に強調表示されている箇所を追加します。 アノテーションは削除します (分かりやすくするためです)。

```
 $ 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 < test; do sleep 1; done
  echo -e "zpm \ninstall samples-bi \nquit \nhalt" > samplesbiinstall
  iris session iris < samplesbiinstall
  rm test samplesbiinstall
  restartPolicy: Always
```

Recreate を使ったアップデート戦略を使用しますが、これはポッドが一旦削除されてから再度作成されることを意味します。 今回はデモなので、これでも大丈夫です。また、使用するリソースも少なくて済みます。
さらに、ポッドが起動すると同時にトリガーする postStart フックも追加しました。 IRIS が起動するのを待ち、デフォルトの zpm-repository から samples-bi をインストールします。
ここで、Kubernetes サービスを追加します (ここでもアノテーションは使いません)。

```
  $ 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
```

そうです、今回は「デフォルト」のネームスペースでデプロイします。デモですからね。

*何*を*どこ*で実行するかが分かったところで、 最後は、*どのように*実行するか、を見ます。

GitHub Actions のワークフロー

すべてを一から作るのではなく、GitHub Actions を使って GKE に InterSystems IRIS Solution をデプロイする に記載されているワークフローに似たものを作成します。 今回は、コンテナを構築する必要はありません。 GKE 特有の箇所は EKS 特有の箇所に置き換えられます。
太文字の箇所は、コミットメッセージを受けて条件付きステップで使用することに関する記述です。

```
  $ cat /.github/workflows/workflow.yaml
name: EKS クラスタをプロビジョンし、そこに Samples BI をデプロイする
on:
  push:
  branches:
  - master
# 環境変数。
# ${{ secrets }} は GitHub -> Settings -> Secrets より取得されます
# ${{ github.sha }} はコミットハッシュです
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: EKS クラスタをプロビジョンする
  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}
```

もちろん、“ terraform ” ユーザーの認証情報を設定 (~/.aws/credentials ファイルから取得) し、Github がそのシークレットを使用することを許可する必要があります。

ワークフローの強調表示されている箇所に注目してください。 これらは、“ [destroy eks] ” というフレーズを持つコミットメッセージをプッシュし、EKS クラスターを破壊することを可能にします。 私たちはそのようなコミットメッセージを持つ “ kubernetes apply ” は実行しません。 パイプラインを実行します。でも最初に .gitignore ファイルを作成します。

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

GitHub のリポジトリページにある「Actions」タブでデプロイプロセスをモニタリングします。 正常に完了するのを待ってください。

ワークフローを最初に実行するときは、“ Terraform apply ” のステップにおよそ 15 分かかります (クラスターの作成にかかる時間とほぼ同じ時間です)。 次に起動するとき (クラスターを削除していなければ) は、ワークフローの実行速度が大幅にアップします。 こちらをチェックしてください。

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

もちろん、作成したワークフローを確認しておくとよいでしょう。 今回は、IAM “ my-user ” の認証情報をご自分のノートパソコンでお使いください。

```
$ 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
```

http://a2c6f6733557511eab3c302618b2fae2-622862917.eu-west-1.elb.amazonaws.com:52773/csp/user/DeepSee.UserPortal.Home.zen?$NAMESPACE=USER
(リンクを自分の External-IP と入れ替えてください) に移動し、“ system ”、“ SYS ”

と順に入力して、デフォルトのパスワードを変更します。 たくさんの BI ダッシュボードが表示されるはずです。

細かく確認するには、それぞれの矢印をクリックします。

samples-bi ポッドを再起動したら、すべての変更内容が消去されるので覚えておきましょう。
これはデモなので、意図的にそうなるようにしています。 永続化する必要がある方のために、例を 1 つ github-gke-zpm-registry/k8s/statefulset.tpl リポジトリに作成しておきました。

作業が終わったら、作成したものをすべて削除してください。
```
  $ git commit -m "Mr Proper [destroy eks]" --allow-empty
$ git push
```

まとめ

この記事では、eksctl ユーティリティの代わりに Terraform を使って EKS クラスターを作成しました。
これで、AWS インフラストラクチャのすべてを「体系化」することに一歩近づけたと思います。
Github Actions と Terraform で git push
使い簡単にデモアプリケーションをデプロイする方法をデモンストレーションしました。
また、ツールボックスに kompose とポッドの postStart フックも追加しました。
今回は、TLS の有効化については触れていません。 それは、また近いうちに取り上げたいと思います。

#AWS #DevOps #Docker #Kubernetes #クラウド #コンテナ化 #InterSystems IRIS #Open Exchange
InterSystems Open Exchangeで関連アプリケーションを確認してください

ソースURL:https://jp.community.intersystems.com/post/github-actions-%E3%82%92%E4%BD%BF%E3%81%A3%E3%81%A6-eks-%E3%81%AB-intersystems-iris-solution-%E3%82%92%E3%83%87%E3%83%97%E3%83%AD%E3%82%A4%E3%81%99%E3%82%8B