Article
Mikhail Khomenko · Mar 12, 2020 23m read

Open Exchange

Deploying an InterSystems IRIS Solution on EKS using GitHub Actions

Imagine you want to see what InterSystems can give you in terms of data analytics. You studied the theory and now you want some practice. Fortunately, InterSystems provides a project that contains some good examples: Samples BI. Start with the README file, skipping anything associated with Docker, and go straight to the step-by-step installation. Launch a virtual instance, install IRIS there, follow the instructions for installing Samples BI, and then impress the boss with beautiful charts and tables. So far so good.

Inevitably, though, you ' ll need to make changes.

It turns out that keeping a virtual machine on your own has some drawbacks, and it ' s better to keep it with a cloud provider. Amazon seems solid, and you create an AWS account (free to start), read that using the root user identity for everyday tasks is evil, and create a regular IAM user with admin permissions.

Clicking a little, you create your own VPC network, subnets, and a virtual EC2 instance, and also add a security group to open the IRIS web port (52773) and ssh port (22) for yourself. Repeat the installation of IRIS and Samples BI. This time, use Bash scripting, or Python if you prefer. Again, impress the boss.

But the ubiquitous DevOps movement leads you to start reading about Infrastructure as Code and you want to implement it. You choose Terraform, since it ' s well-known to everyone and its approach is quite universal—suitable with minor adjustments for various cloud providers. You describe the infrastructure in HCL language, and translate the installation steps for IRIS and Samples BI to Ansible. Then you create one more IAM user to enable Terraform to work. Run it all. Get a bonus at work.

Gradually you come to the conclusion that in our age of microservices it ' s a shame not to use Docker, especially since InterSystems tells you how. You return to the Samples BI installation guide and read the lines about Docker, which don ' t seem to be complicated:

```
$ 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
```

After directing your browser to http://localhost:52773/csp/user/DeepSee.UserPortal.Home.zen?$NAMESPACE=USER, you again go to the boss and get a day off for a nice job.

You then begin to understand that " docker run " is just the beginning, and you need to use at least docker-compose. Not a problem:

```
$ 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
```

So you install Docker and docker-compose with Ansible, and then just run the container, which will download an image if it ' s not already present on the machine. Then you install Samples BI.

You certainly like Docker, because it ' s a cool and simple interface to various kernel stuff. You start using Docker elsewhere and often launch more than one container. And find that often containers must communicate with each other, which leads to reading about how to manage multiple containers.

And you come to Kubernetes.

One option to quickly switch from docker-compose to Kubernetes is to use kompose. Personally, I prefer to simply copy Kubernetes manifests from manuals and then edit for myself, but kompose does a good job of completing its small task:

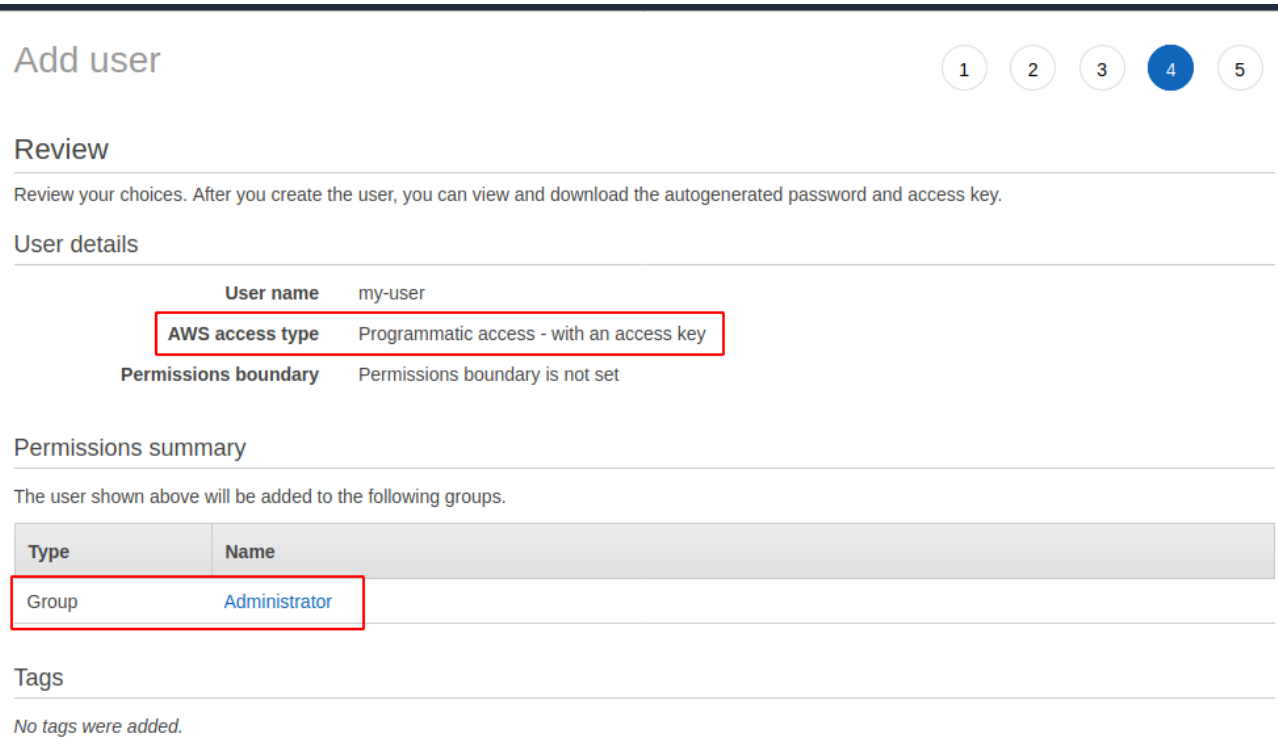
```
$ kompose convert -f docker-compose.yml
INFO Kubernetes file "irisce-service.yaml" created
INFO Kubernetes file "irisce-deployment.yaml" created
```

Now you have the deployment and service files that can be sent to some Kubernetes cluster. You find out that you can install a minikube, which lets you run a single-node Kubernetes cluster and is just what you need at this stage. After a day or two of playing with the minikube sandbox, you ' re ready to use a real live Kubernetes deployment somewhere in the AWS cloud.

Getting Set Up

So, let ' s do this together. At this point we'll make a couple assumptions:

First, we assume you have an AWS account, you know its ID, and you don ' t use root credentials. You create an IAM user (let's call it " my-user ") with administrator rights and programmatic access only and store its credentials. You also create another IAM user, called " terraform, " with the same permissions:



On its behalf, Terraform will go to your AWS account and create and delete the necessary resources. The extensive rights of both users are explained by the fact that this is a demo. You save credentials locally for both IAM users:

```
$ 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
```

Note: Don ' t copy and paste the credentials from above. They are provided here as an example and no longer exist. Edit the ~/.aws/credentials file and introduce your own records.

Second, we ' ll use the dummy AWS Account ID (01234567890) for the article, and the AWS region " eu-west-1. " Feel free to use another region.

Third, we assume you ' re aware that AWS is not free and you ' ll have to pay for resources used.

Next, you ' ve installed the AWS CLI utility for command-line communication with AWS. You can try to use aws2, but you ' ll need to specifically set aws2 usage in your kube config file, as described here.

You ' ve also installed the kubectl utility for command-line communication with AWS Kubernetes.

And you ' ve installed the kompose utility for docker-compose.yml for converting Kubernetes manifests.

Finally, you ' ve created an empty GitHub repository and cloned it to your host. We ' ll refer to its root directory as <root_repo_dir>. In this repository, we ' ll create and fill three directories: .github/workflows/, k8s/, and terraform/.

Note that all the relevant code is duplicated in the github-eks-samples-bi repo to simplify copying and pasting.

Let ' s continue.

AWS EKS Provisioning

We already met EKS in the article Deploying a Simple IRIS-Based Web Application Using Amazon EKS. At that time, we created a cluster semi-automatically. That is, we described the cluster in a file, and then manually launched the eksctl utility from a local machine, which created the cluster according to our description.

eksctl was developed for creating EKS clusters and it ' s good for a proof-of-concept implementation, but for everyday usage it ' s better to use something more universal, such as Terraform. A great resource, AWS EKS Introduction, explains the Terraform configuration needed to create an EKS cluster. An hour or two spent getting acquainted with it will not be a waste of time.

You can play with Terraform locally. To do so, you ' ll need a binary (we ' ll use the latest version for Linux at the time of writing of the article, 0.12.20), and the IAM user " terraform " with sufficient rights for Terraform to go to AWS. Create the directory <rootrepodir>/terraform/ to store Terraform code:

```
$ mkdir <rootrepodir>/terraform
$ cd <rootrepodir>/terraform
```

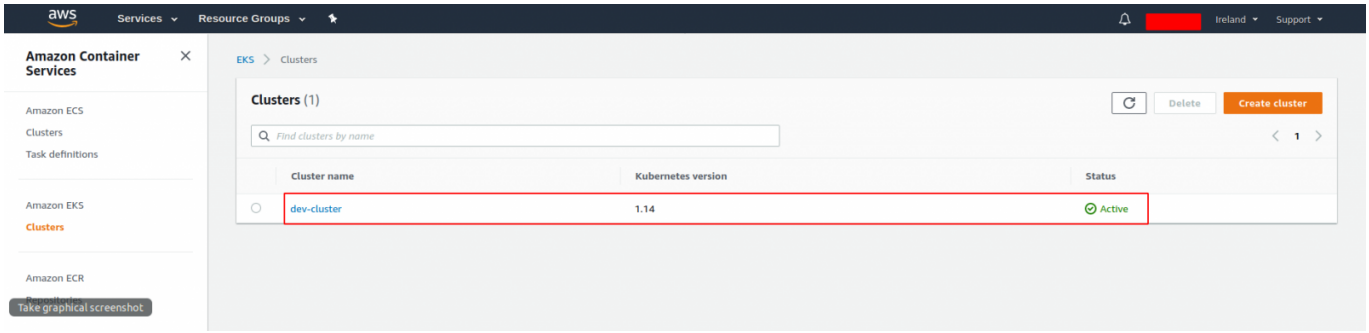
You can create one or more *.tf* files (they are merged at startup). Just copy and paste the code examples from AWS EKS Introduction and then run something like:

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

You may encounter some errors. If so, play a little with debug mode, but remember to turn it off later:

```
$ export TFLOG=debug
$ terraform plan -out eks.plan
<many-many lines here>
$ unset TFLOG
```

This experience will be useful, and most likely you ' ll get an EKS cluster launched (use " terraform apply " for that). Check it out in the AWS console:



Clean up when you get bored:

```
$ terraform destroy
```

Then go to the next level and start using the Terraform EKS module, especially since it's based on the same EKS introduction. In the examples/ directory you'll see how to use it. You'll also find other examples there.

We simplified the examples somewhat. Here's the main file in which the VPC creation and EKS creation modules are called:

```
$ cat <rootrepodir>/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.awseksclusterauth.cluster.endpoint
  clustercacertificate =
base64decode(data.awseksclusterauth.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 "awseksclusterauth" "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
}
```

Let ' s look a little more closely at the *"terraform"* block in 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"
  }
}
```

Here we indicate that we ' ll adhere to the syntax not lower than Terraform 0.12 (much has changed compared with earlier versions), and also that Terraform shouldn ' t store its state locally, but rather remotely, in the S3 bucket.

It ' s convenient if the terraform code can be updated from different places by different people, which means we need to be able to lock a user ' s state, so we added a lock using a dynamodb table. Read more about locks on the State Locking page.

Since the name of the bucket should be unique throughout AWS, the name " eks-github-actions-terraform " won ' t work for you. Please think up your own and make sure it ' s not already taken (so you ' re getting a *NoSuchBucket* error):

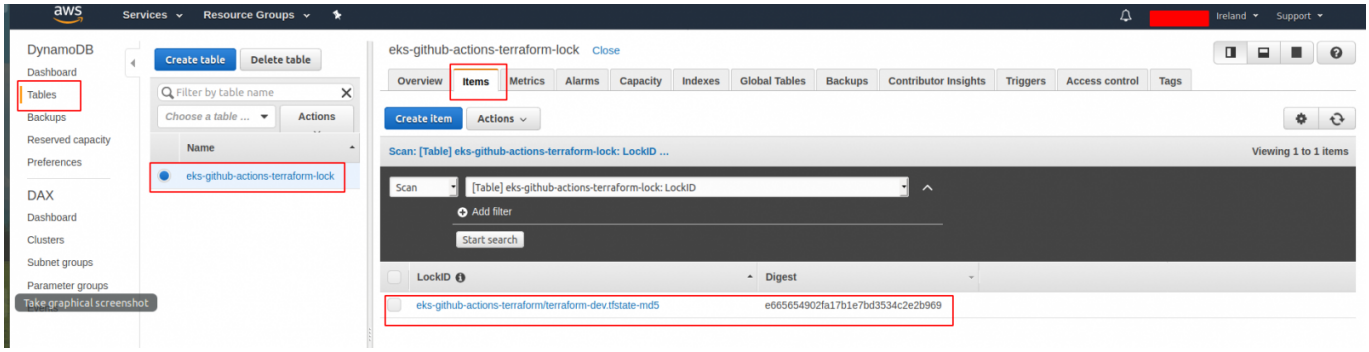
```
$ aws s3 ls s3://my-bucket
An error occurred (AllAccessDisabled) when calling the ListObjectsV2 operation:
All access to this object has been disabled
$ aws s3 ls s3://my-bucket-with-name-that-impossible-to-remember
An error occurred (NoSuchBucket) when calling the ListObjectsV2 operation: The
specified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

Having come up with a name, create the bucket (we use the IAM user “ terraform ” here. It has administrator rights so it can create a bucket) and enable versioning for it (which will save your nerves in the event of a configuration error):

```
$ aws s3 mb s3://eks-github-actions-terraform --region eu-west-1
make_bucket: 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"
}
```

With DynamoDB, uniqueness is not needed, but you do need to create a table first:

```
$ 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
```



Keep in mind that, in case of Terraform failure, you may need to remove a lock manually from the AWS console. But be careful when doing so.

With regard to the module eks/vpc blocks in main.tf, the way to reference the module available on GitHub is simple:

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

Now let ’ s look at our other two Terraform files (variables.tf and outputs.tf). The first holds our Terraform variables:

```
$ cat <root_repo_dir>/terraform/variables.tf
variable "region" {
  default = "eu-west-1"
}
variable "map_accounts" {
```

```
  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"]
  }
  ]
}
```

The most important part here is adding the IAM user “ my-user ” to the mapusers variable, but you should use your own account ID here in place of 01234567890.

What does this do? When you communicate with EKS through the local kubectl client, it sends requests to the Kubernetes API server, and each request goes through authentication and authorization processes so Kubernetes can understand who sent the request and what they can do. So the EKS version of Kubernetes asks AWS IAM for help with user authentication. If the user who sent the request is listed in AWS IAM (we pointed to his ARN here), the request goes to the authorization stage, which EKS processes itself, but according to our settings. Here, we indicated that the IAM user “ my-user ” is very cool (group “ system: masters ”).

Finally, the outputs.tf file describes what Terraform should print after it finishes a job:

```
$ cat <rootrepodir>/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
}
```

This completes the description of the Terraform part. We ' ll return soon to see how we ' re going to launch these files.

Kubernetes Manifests

So far, we ' ve taken care of *where* to launch the application. Now let ' s look at *what* to run.

Recall that we have docker-compose.yml (we renamed the service and added a couple of labels that kompose will use shortly) in the <rootrepodir>/k8s/ directory:

```
$ cat <rootrepodir>/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
```

Run kompose and then add what ' s highlighted below. Delete annotations (to make things more intelligible):

```
$ kompose convert -f docker-compose.yml --replicas=1
$ cat <rootrepodir>/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
```

We use the Recreate update strategy, which means that the pod will be deleted first and then recreated. This is permissible for demo purposes and allows us to use fewer resources.
We also added the postStart hook, which will trigger immediately after the pod starts. We wait until IRIS starts up and install the samples-bi package from the default zpm-repository.
Now we add the Kubernetes service (also without annotations):

```
$ cat <rootrepodir>/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
```

Yes, we ' ll deploy in the " default " namespace, which will work for the demo.

Okay, now we know *where* and *what* we want to run. It remains to see *how*.

The GitHub Actions Workflow

Rather than doing everything from scratch, we ' ll create a workflow similar to the one described in Deploying InterSystems IRIS solution on GKE Using GitHub Actions. This time we don ' t have to worry about building a container. The GKE-specific parts are replaced by those specific to EKS. Bolded parts are related to receiving the commit message and using it in conditional steps:

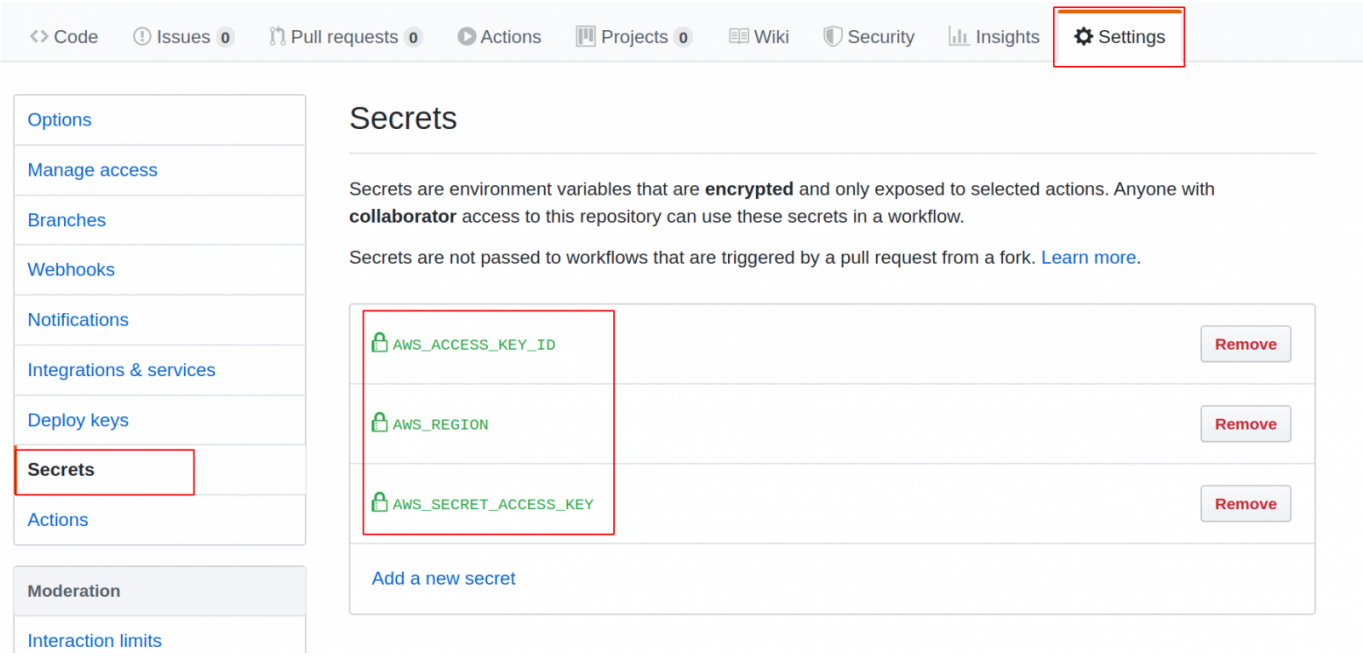
```
$ cat <rootrepodir>/.github/workflows/workflow.yaml
name: Provision EKS cluster and deploy Samples BI there
on:
  push:
  branches:
  - master
# Environment variables.
# ${{ secrets }} are taken from GitHub -> Settings -> Secrets
# ${{ github.sha }} is the commit hash
env:
  AWSACCESSKEYID: ${{ secrets.AWSACCESSKEYID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

Of course, we need to set the credentials of the " terraform " user (take them from the ~/.aws/credentials file), letting Github use its secrets:



Notice the highlighted parts of workflow. They will enable us to destroy an EKS cluster by pushing a commit message that contains a phrase " [destroy eks] " . Note that we won ' t run " kubernetes apply " with such a commit message.
Run a pipeline, but first create a .gitignore file:

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

Monitor deployment process on the "Actions" tab of GitHub repository page. Please wait for successful completion.

When you run a workflow for the very first time, it will take about 15 minutes on the " Terraform apply " step, approximately as long as it takes to create the cluster. At the next start (if you didn ' t delete the cluster), the workflow will be much faster. You can check this out:

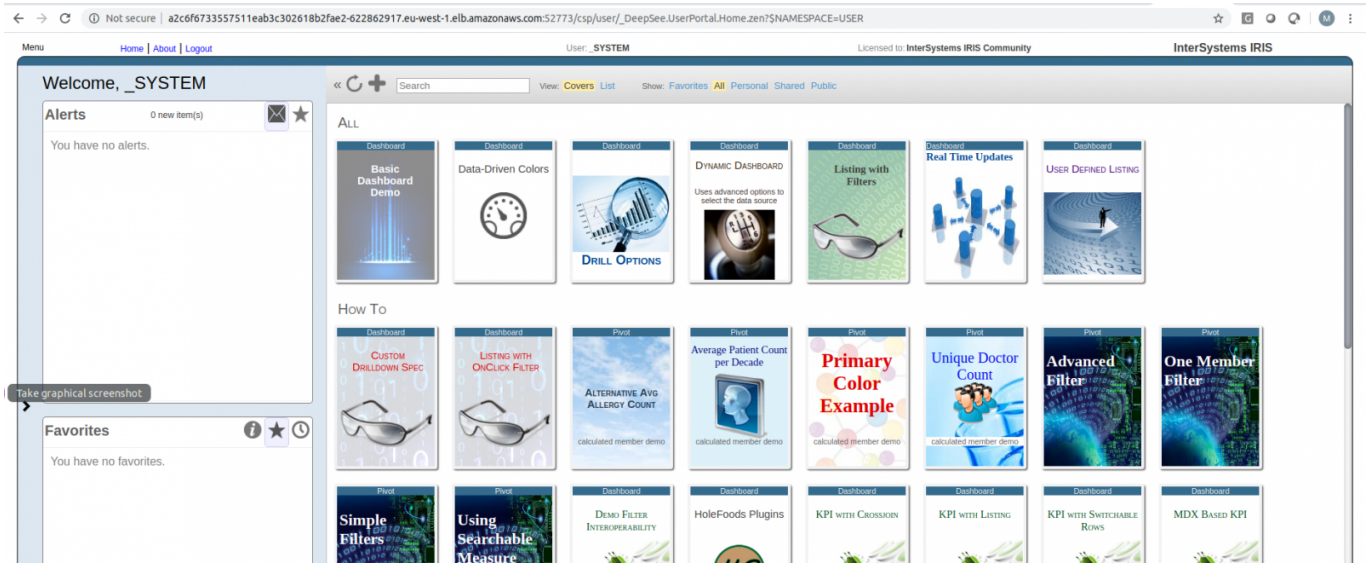
```
$ cd <rootrepodir>
$ git commit -m "Trigger" --allow-empty
```

```
$ git push
```

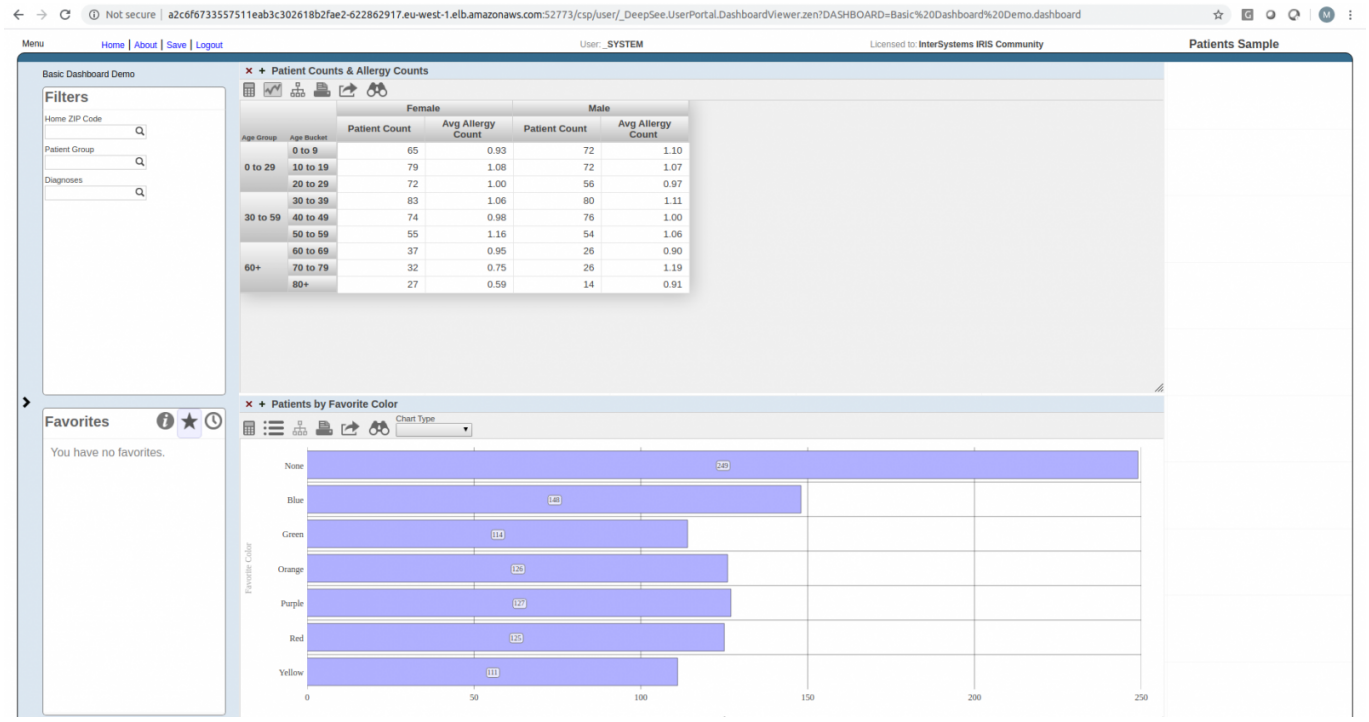
Of course, it would be nice to check what we did. This time you can use the credentials of IAM " my-user " on your laptop:

```
$ export AWS_PROFILE=my-user
$ export AWS_REGION=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 <none> 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 <none> 443/TCP 11m
samples-bi LoadBalancer 172.20.33.235
a2c6f6733557511eab3c302618b2fae2-622862917.eu-west-1.elb.amazonaws.com 52773:31047/TCP 6m33s
```

Go to http://a2c6f6733557511eab3c302618b2fae2-622862917.eu-west-1.elb.amazonaws.com:52773/csp/user/DeepSee.UserPortal.Home.zen?$NAMESPACE=USER
(substitute link by your External-IP), then type " system ", " SYS " and change the default password. You should see a bunch of BI dashboards:



Click on each one ' s arrow to deep dive:



Remember, if you restart a samples-bi pod, all your changes will be lost. This is intentional behavior as this is a demo. If you need persistence, I've created an example in the github-gke-zpm-registry/k8s/statefulset.tpl repository.

When you ' re finished, just remove everything you ' ve created:

```
$ git commit -m "Mr Proper [destroy eks]" --allow-empty
$ git push
```

Conclusion

In this article, we replaced the eksctl utility with Terraform to create an EKS cluster. It ' s a step forward to " codify " all

of your AWS infrastructure.
We showed how you can easily deploy a demo application with git push using Github Actions and Terraform.
We also added kompose and a pod ' s postStart hooks to our toolbox.
We didn ' t show TLS enabling this time. That ' s a task we ' ll undertake in the near future.

#AWS #Best Practices #Cloud #Containerization #DevOps #Docker #Kubernetes #InterSystems IRIS #Open Exchange
Check the related application on InterSystems Open Exchange

Source URL:https://community.intersystems.com/post/deploying-intersystems-iris-solution-eks-using-github-actions