Article
Mikhail Khomenko · Dec 23, 2019 12m read

Open Exchange

Deploying a Simple IRIS-Based Web Application Using Amazon EKS

Last time we deployed a simple IRIS application to the Google Cloud. Now we ' re going to deploy the same project to Amazon Web Services using its Elastic Kubernetes Service (EKS).

We assume you ' ve already forked the IRIS project to your own private repository. It ' s called *<username>/my-objectscript-rest-docker-template* in this article. *<rootrepodir>* is its root directory.

Before getting started, install the AWS command-line interface and, for Kubernetes cluster creation, eksctl, a simple CLI utility. For AWS you can try to use aws2, but you ' ll need to set aws2 usage in kube config file as described here.

AWS EKS

Like AWS resources in general, EKS is not free. But you can create a free-tier account to play with AWS features. Keep in mind, though, that not everything you want to play with is included in the free tier. So, to manage your current budget and understand the financial issues, read this and this.

We ' ll assume you already have an AWS account and root access to it, and that you don ' t use this root access but have created a user with admin permissions. You ' ll need to put the access key and secret key of this user into the AWS credentials file under the [dev] profile (or whatever you choose to name the profile):

```
$ cat ~/.aws/credentials
[dev]
aws_access_key_id = ABCDEFGHIJKLMNOPQRST
aws_secret_access_key =
1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ1234
```

We ' re going to create resources in the AWS " eu-west-1 " region, but you should choose the region closest to your location and replace " eu-west-1 " by your region everywhere below in the text.

By the way, all needed files (.circleci/, eks/, k8s/) are also stored here to simplify copying and pasting.

All required EKS resources will be created from scratch. You ' ll find the Amazon EKS Workshop to be a good resource to get an initial impression.

Now let ' s check our access to AWS (we ' ve used a dummy account here):

```
$ export AWS_PROFILE=dev

$ aws sts get-caller-identity
{
  "Account": "012345678910",
  "UserId": " ABCDEFGHIJKLMNOPQRSTU",
  "Arn": "arn:aws:iam::012345678910:user/FirstName.LastName"
```

```
}


$ eksctl version
[    ] version.Info{BuiltAt:"", GitCommit:"", GitTag:"0.10.2"}
```

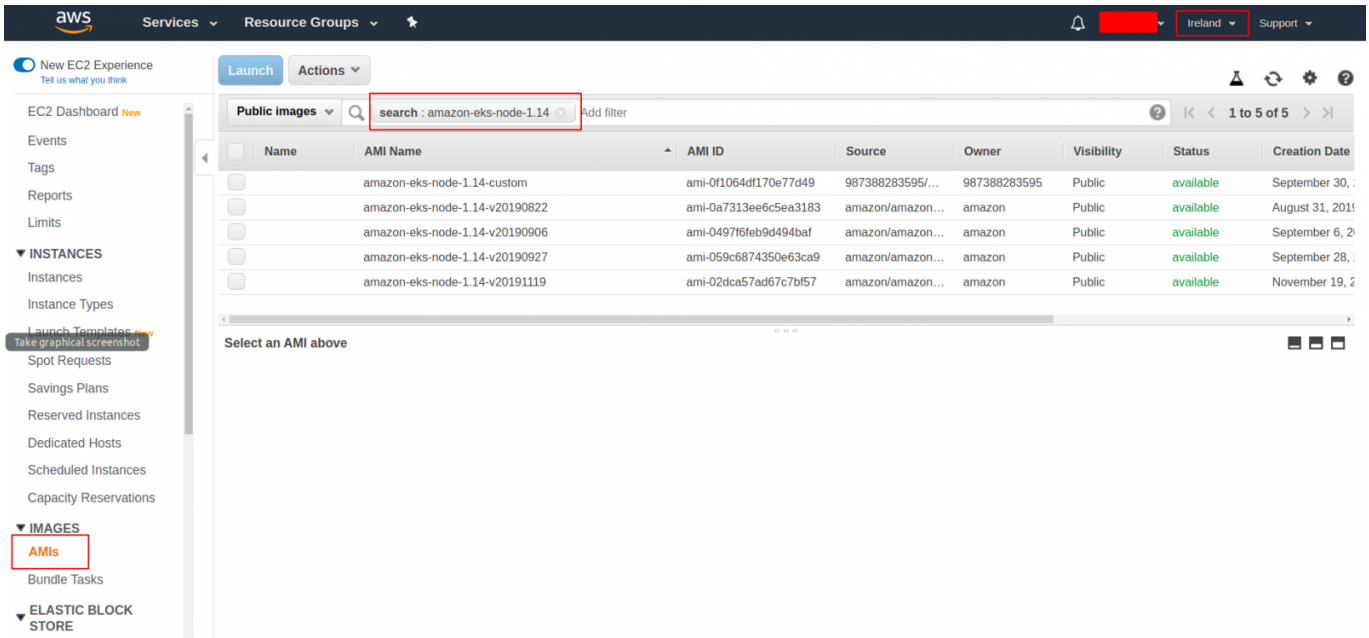
We could run *"eksctl create cluster <clustername> --region eu-west-1"* now, relying on the fact that all the default settings are good for us, or we can manage our own settings by creating a configuration file and using it.

The latter is preferable because it allows you to store such a file in a version control system (VCS). Examples of configurations can be found here. After reading about the different settings here, let's try to create our own configuration:

```
$ mkdir <rootrepodir>/eks; cd <rootrepodir>/eks

$ cat cluster.yaml

apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dev-cluster
  region: eu-west-1
  version: '1.14'

vpc:
  cidr: 10.42.0.0/16
  nat:
  gateway: Single
  clusterEndpoints:
  publicAccess: true
  privateAccess: false

nodeGroups:
  - name: ng-1
  amiFamily: AmazonLinux2
  ami: ami-059c6874350e63ca9 # AMI is specific for a region
  instanceType: t2.medium
  desiredCapacity: 1
  minSize: 1
  maxSize: 1

  # Worker nodes won't have an access FROM the Internet
  # But have an access TO the Internet through NAT-gateway
  privateNetworking: true
```

```
  # We don't need to SSH to nodes for demo
  ssh:
  allow: false

  # Labels are Kubernetes labels, shown when 'kubectl get nodes --show-labels'
  labels:
  role: eks-demo
  # Tags are AWS tags, shown in 'Tags' tab on AWS console'
  tags:
  role: eks-demo

# CloudWatch logging is disabled by default to save money
# Mentioned here just to show a way to manage it
#cloudWatch:
# clusterLogging:
# enableTypes: []
```

Note that "*nodeGroups.desiredCapacity = 1*" would make no sense in a production environment, but it ' s fine for our demo.
Also note that AMI images could differ between regions. Look for "amazon-eks-node-1.14" and choose one of the latest:



Now let ' s create the cluster—the control plane and worker nodes:

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

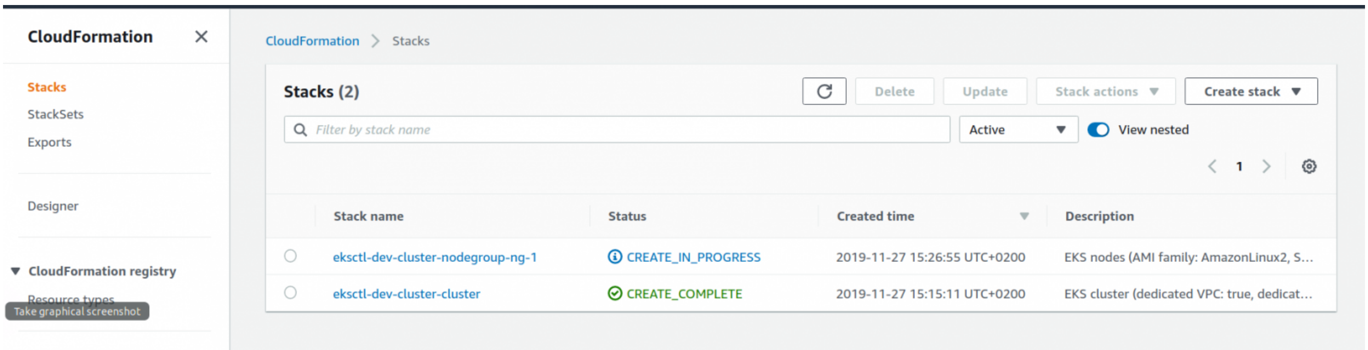
By the way, when you no longer need a cluster, you can use the following to delete it:

$ eksctl delete cluster --name dev-cluster --region eu-west-1 --wait

Creating a cluster takes about 15 minutes. During this time you can look at the eksctl output:

```
[redacted]eks$ eksctl create cluster -f cluster.yaml
[i]  eksctl version 0.10.2
[i]  using region eu-west-1
[i]  setting availability zones to [eu-west-1c eu-west-1b eu-west-1a]
[i]  subnets for eu-west-1c - public:10.42.0.0/19 private:10.42.96.0/19
[i]  subnets for eu-west-1b - public:10.42.32.0/19 private:10.42.128.0/19
[i]  subnets for eu-west-1a - public:10.42.64.0/19 private:10.42.160.0/19
[i]  nodegroup "ng-1" will use "ami-059c6874350e63ca9" [AmazonLinux2/1.14]
[i]  using Kubernetes version 1.14
[i]  creating EKS cluster "dev-cluster" in "eu-west-1" region
[i]  1 nodegroup (ng-1) was included (based on the include/exclude rules)
[i]  will create a CloudFormation stack for cluster itself and 1 nodegroup stack(s)
[i]  will create a CloudFormation stack for cluster itself and 0 managed nodegroup stack(s)
[i]  if you encounter any issues, check CloudFormation console or try 'eksctl utils describe-stacks --region=eu-west-1 --cluster=dev-cluster'
[i]  CloudWatch logging will not be enabled for cluster "dev-cluster" in "eu-west-1"
[i]  you can enable it with 'eksctl utils update-cluster-logging --region=eu-west-1 --cluster=dev-cluster'
[i]  Kubernetes API endpoint access will use default of {publicAccess=true, privateAccess=false} for cluster "dev-cluster" in "eu-west-1"
[i]  2 sequential tasks: { create cluster control plane "dev-cluster", create nodegroup "ng-1" }
[i]  building cluster stack "eksctl-dev-cluster-cluster"
[i]  deploying stack "eksctl-dev-cluster-cluster"
[i]  building nodegroup stack "eksctl-dev-cluster-nodegroup-ng-1"
[i Take graphical screenshot eksctl-dev-cluster-nodegroup-ng-1"
[✔]  all EKS cluster resources for "dev-cluster" have been created
[✔]  saved kubeconfig as "/home/no-name-pc/.kube/config"
[i]  adding identity "arn:aws:iam::[redacted]:role/eksctl-dev-cluster-nodegroup-ng-1-NodeInstanceRole-FNE1UXTB4TW4" to auth ConfigMap
[i]  nodegroup "ng-1" has 0 node(s)
[i]  waiting for at least 1 node(s) to become ready in "ng-1"
[i]  nodegroup "ng-1" has 1 node(s)
[i]  node "ip-10-42-140-98.eu-west-1.compute.internal" is ready
[✖]  unable to use kubectl with the EKS cluster (check 'kubectl version'): usage: aws [options] <command> <subcommand> [<subcommand> ...] [parameters]
To see help text, you can run:

  aws help
  aws <command> help
  aws <command> <subcommand> help
aws: error: argument operation: Invalid choice, valid choices are:

create-cluster                           | delete-cluster
describe-cluster                         | describe-update
list-clusters                            | list-updates
update-cluster-version                   | update-kubeconfig
wait                                     | help
Unable to connect to the server: getting credentials: exec: exit status 2

[i]  cluster should be functional despite missing (or misconfigured) client binaries
[✔]  EKS cluster "dev-cluster" in "eu-west-1" region is ready
[redacted]eks$
```

You can also view [the CloudFormation console](), which will have two stacks. You can drill down into each one and look at the Resources tab to see exactly what will be created, and at the Events tab to check the current state of the resources creation.

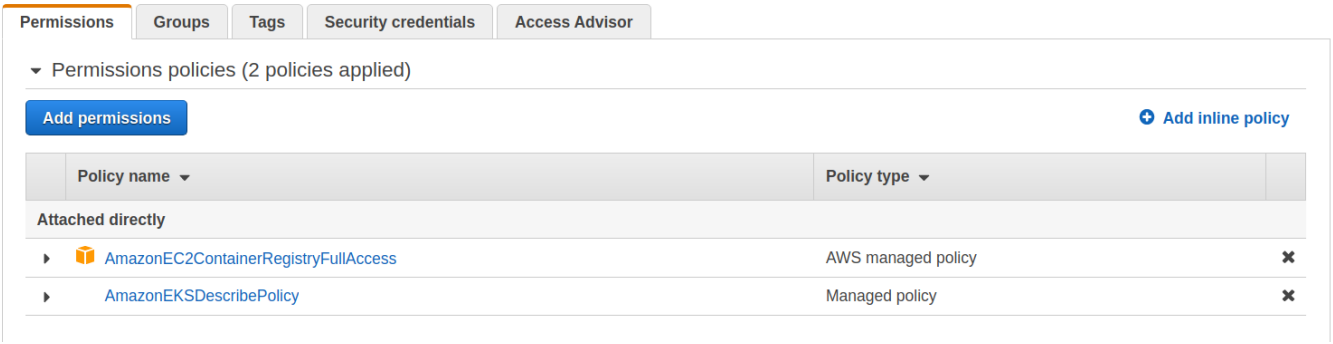


The cluster was successfully created, although you can see in the eksctl output that we had difficulties connecting to it: "unable to use kubectl with the EKS cluster".

Let's fix this by installing the aws-iam-authenticator (IAM) and using it to create a kube context:

```
$ which aws-iam-authenticator
/usr/local/bin/aws-iam-authenticator

$ aws eks update-kubeconfig --name dev-cluster --region eu-west-1

$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
ip-10-42-140-98.eu-west-1.compute.internal Ready <none> 1m
v1.14.7-eks-1861c5
```

It should work now, but we created a cluster with a user who has administrator rights. For the regular deployment process from CircleCI, it ' s better to create a special AWS user, named, in this case, CircleCI, with only programmatic access and the following policies attached:



The first policy is built into AWS, so you only need to select it. The second one should be created on your own. Here is a creation process description. The policy *"AmazonEKSDescribePolicy"* should look like:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "eks:DescribeCluster",
            "eks:ListClusters"
         ],
         "Resource": "*"
      }
   ]
}
```

After user creation, save the user ' s access key and secret access key — we ' ll need them soon.

We also want to give this user rights within the Kubernetes cluster itself, as described in this article. In short, after creating the EKS cluster, only the IAM user, *creator*, has access to it. To add our CircleCI user, we need to replace default empty "mapUsers" section in the cluster AWS authentication settings (configmap aws-auth, 'data' section) by the following lines using kubectl edit (use your own Account Id instead of ' 01234567890 '):

```
$ kubectl -n kube-system edit configmap aws-auth
...
data:
...
  mapUsers: |
  - userarn: arn:aws:iam::01234567890:user/CircleCI
  username: circle-ci
  groups:
  - system:masters
```

We ' ll use the Kubernetes manifests from the earlier article (see the " Google Cloud Prerequisites " section) with one change: in the deployment image field we use placeholders. We ' ll store those manifests in the *<rootrepodir>/k8s* directory. Note that the deployment file was renamed to *deployment.tpl*:

```
$ cat <rootrepodir>/k8s/deployment.tpl
...
spec:
containers:
- image: DOCKERREPONAME/iris-rest:DOCKERIMAGETAG
...
```

CircleCI

The deployment process on the CircleCI side is similar to the process used for GKE:

- Pull the repository
- Build the Docker image
- Authenticate with Amazon Cloud
- Upload the image to Amazon Elastic Container Registry (ECR)
- Run the container based on this image in AWS EKS

Like last time, we ' ll take advantage of already created and tested CircleCI configuration templates orbs.

- aws-ecr orb for building an image and pushing it to ECR
- aws-eks orb for AWS authentication
- kubernetes orb for Kubernetes manifests deployment

Our deployment configuration looks like this:

```
$ cat <rootrepodir>/.circleci/config.yml
version: 2.1
orbs:
```

```
  aws-ecr: circleci/aws-ecr@6.5.0
  aws-eks: circleci/aws-eks@0.2.6
  kubernetes: circleci/kubernetes@0.10.1
jobs:
  deploy-application:
  executor: aws-eks/python3
  parameters:
  cluster-name:
  description: ¦
  Name of the EKS cluster
  type: string
  aws-region:
  description: ¦
  AWS region
  type: string
  account-url:
  description: ¦
  Docker AWS ECR repository url
  type: string
  tag:
  description: ¦
  Docker image tag
  type: string
  steps:
  - checkout
  - run:
  name: Replace placeholders with values in deployment template
  command: ¦
  cat k8s/deployment.tpl ¦\
  sed "s¦DOCKERREPONAME¦<< parameters.account-url >>¦" ¦\
  sed "s¦DOCKERIMAGETAG¦<< parameters.tag >>¦" > k8s/deployment.yaml; \
  cat k8s/deployment.yaml
  - aws-eks/update-kubeconfig-with-authenticator:
  cluster-name: << parameters.cluster-name >>
  install-kubectl: true
  aws-region: << parameters.aws-region >>
  - kubernetes/create-or-update-resource:
  action-type: apply
  resource-file-path: "k8s/namespace.yaml"
  show-kubectl-command: true
  - kubernetes/create-or-update-resource:
  action-type: apply
```

```
  resource-file-path: "k8s/deployment.yaml"
  show-kubectl-command: true
  get-rollout-status: true
  resource-name: deployment/iris-rest
  namespace: iris
  - kubernetes/create-or-update-resource:
  action-type: apply
  resource-file-path: "k8s/service.yaml"
  show-kubectl-command: true
  namespace: iris
workflows:
  main:
  jobs:
  - aws-ecr/build-and-push-image:
  aws-access-key-id: AWSACCESSKEYID
  aws-secret-access-key: AWSSECRETACCESSKEY
  region: AWSREGION
  account-url: AWSECRACCOUNTURL
  repo: iris-rest
  create-repo: true
  dockerfile: Dockerfile-zpm
  path: .
  tag: ${CIRCLESHA1}
  - deploy-application:
  cluster-name: dev-cluster
  aws-region: eu-west-1
  account-url: ${AWSECRACCOUNTURL}
  tag: ${CIRCLESHA1}
  requires:
  - aws-ecr/build-and-push-image
```

The Workflows section contains a list of jobs, each of which can be either called from an orb, such as aws-ecr/build-and-push-image, or defined directly in the configuration using “ deploy-application ” .

The following code means that the deploy-application job will be called only after the aws-ecr/build-and-push-image job finishes:

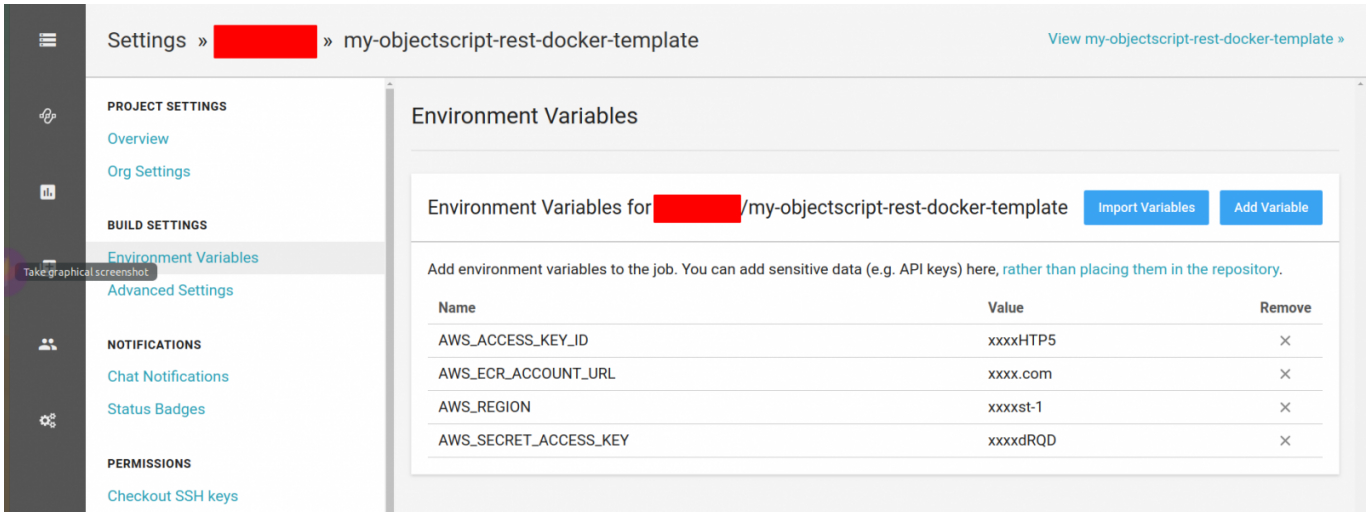
```
requires:
- aws-ecr/build-and-push-image
```

The Jobs section contains a description of the deploy-application job, with a list of steps defined, including:

- *checkout*, to pull from a Git repository

- *run*, to run a script that dynamically sets the Docker-image repository and tag
- *aws-eks/update-kubeconfig-with-authenticator*, which uses aws-iam-authenticator to set up a connection to Kubernetes
- kubernetes/create-or-update-resource, which is used several times as a way to run " kubectl apply " from CircleCI

We use variables and they, of course, should be defined in CircleCI on the " Environment variables " tab:



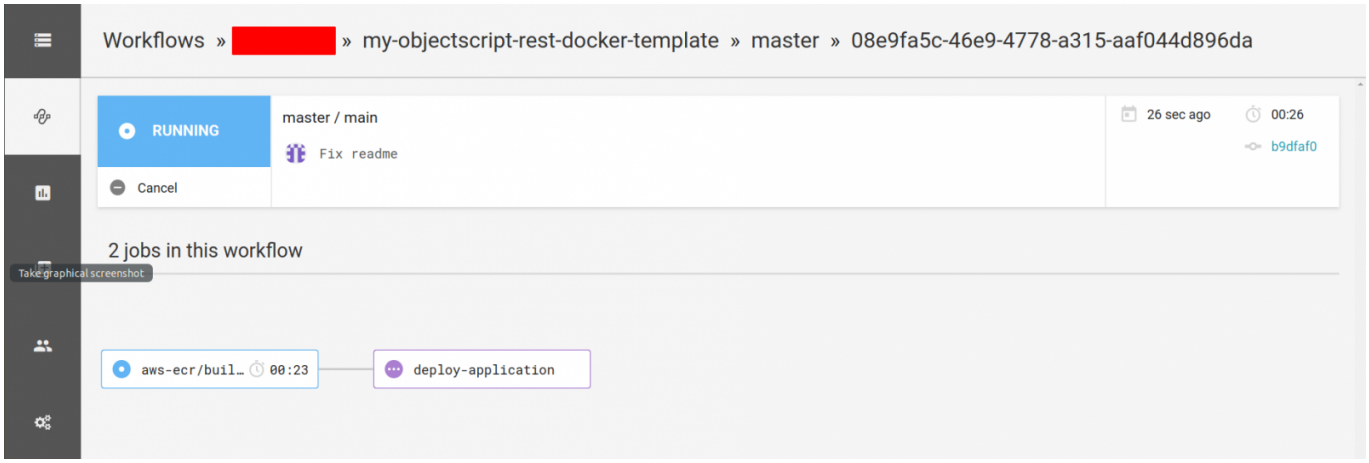
The following table shows the meaning of the variables used:

AWSACCESSKEYID	Access key of CircleCI
AWSSECRETACCESSKEY	Secret key of CircleCI
AWSREGION	*eu-west-1*, in this case
AWSECRACCOUNTURL	URL of the AWS ECR as *01234567890.dkr.ec* where ' 01234567890

Here ' s how we trigger the deployment process:

```
$ git add .circleci/ eks/ k8s/
$ git commit -m " AWS EKS deployment "
$ git push
```

This will show the two jobs in this workflow:



Both jobs are clickable, and this allows you to see details of the steps taken.
Deployment takes several minutes. Once it completes, we can check the status of the Kubernetes resources and of the IRIS application itself:

```
$ kubectl -n iris get pods -w   # Ctrl+C to stop

$ kubectl -n iris get service
NAME  TYPE  CLUSTER-IP  EXTERNAL-IP  PORT(S)  AGE
iris-rest LoadBalancer 172.20.190.211  a3de52988147a11eaaaff02ca6b647c2-66
3499201.eu-west-1.elb.amazonaws.com 52773:32573/TCP 15s
```

Allow several minutes to propagate the DNS-record. Until then you ' ll receive a " Could not resolve host " error when running curl:

```
$ curl -XPOST -H "Content-Type: application/json" -u system:SYS a3de5298814
7a11eaaaff02ca6b647c2-663499201.eu-
west-1.elb.amazonaws.com:52773/person/ -d '{"Name":"John Dou"}'

$ curl -XGET -u system:SYS a3de52988147a11eaaaff02ca6b647c2-663499201.
eu-west-1.elb.amazonaws.com:52773/person/all
[{"Name":"John Dou"},]
```

Wrapping up

At first glance, deployment to AWS EKS looks more complex than to GKE, but it ' s not really much different. If your organization uses AWS, you now know how to add Kubernetes to your stack.

Note that the EKS API was recently extended to support managed groups. These allow you to deploy the control plane and the data plane as a whole, and they look promising. Moreover, Fargate, the AWS serverless compute engine for containers, is now available.

Finally, a quick note about AWS ECR: don ' t forget to set up a lifecycle policy for your images.

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