Create an Amazon Elastic Kubernetes Service (EKS) cluster with jx

In the lesson we will discuss how to create a EKS cluster with jx.

We'll cover the following

Setting environment variables

To interact with AWS through its CLI, we need environment variables

To interact with AWS through its CLI, we need environment variables

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_DEFAULT_REGION`

There are other ways, but I'll ignore them in this course.

```
export AWS_ACCESS_KEY_ID=[...]

export AWS_SECRET_ACCESS_KEY=[...]

export AWS_DEFAULT_REGION=us-east-1
```

Please replace the first `[...]` with the AWS Access Key ID, and the second with the AWS Secret Access Key. I am assuming that you are already familiar with AWS and you know how to create those keys, or that you already have them. If that's not the case, please follow the instructions from the Managing Access Keys for Your AWS Account Root User page.

Creating an Amazon Elastic Kubernetes Service (EKS) cluster with `jx`

Now we're ready to create an EKS cluster.

Cluster name, region and node type

We'll name it `jx-rocks` (`--cluster-name`). It will run inside `us-east-1` region (`--region`) and on `t2.large` (2 CPUs and 8 GB RAM) machines (`--node-type`).

Cluster autoscaling

Unlike with GKE, we won't get a Cluster Autoscaler out of the box, but we'll fix that later. For now, you can assume that there will eventually be autoscaling, so there's no need to worry whether the current capacity is enough. If anything, it is likely more than we will need from the start. Still, even though autoscaling will come later, we'll set the current (`--nodes`) and minimum (`--nodes-min`) number of nodes to three and the maximum to six (`--nodes-max`). That will be converted into AWS Auto-Scaling Groups and, in the case of a misstep, it'll protect us from ending up with more nodes than we can afford.

Default admin password

We'll also set the default Jenkins X password to `admin` (`--default-admin-password`)

Otherwise, the process will create a random one.

Default environment prefix

Finally, we'll set `jx-rocks` as the default environment prefix (`--default-environment-prefix`). A part of the process will create a few repositories (one for staging and the other for production), and that prefix will be used to form their names. We won't go into much detail about those environments and repositories just yet, that's reserved for one of the follow-up chapters.

Feel free to change any of the values in the command that follows to suit your needs better. Or, keep them as they are. After all, this is only a practice; you'll be able to destroy the cluster and recreate it later on with different values.

```
jx create cluster eks \
    --cluster-name jx-rocks \
    --region $AWS_DEFAULT_REGION \
    --node-type t2.large \
    --nodes 3 \
    --nodes-min 3 \
    --nodes-max 6 \
    --default-admin-password admin \
    --default-environment-prefix jx-rocks
```

What do we get from this command?

Let's explore what we're getting from this command. You should be able to correlate my explanation with the console output.

> ⚠ If you get stuck with the `waiting for external loadbalancer to be created and update the nginx-ingress-controller service in kube-system namespace`, you probably encountered a bug. To fix it, open the AWS console and remove the `kubernetes.io/cluster/jx-rocks` tag from the security group `eks-cluster-sg-*`.

> ⚠ Do not be too hasty answering `jx` questions. For all other types of Kubernetes clusters, we can safely use the default answers (enter key). But, in the case of EKS, there is one question that we'll answer with a non-default value. I'll explain it in more detail when we get there. For now, keep an eye on the "`would you like to register a wildcard DNS ALIAS to point at this`

> `ELB address?` ” question.

The process started creating an EKS cluster right away. This will typically take around ten minutes, during which you won't see any movement in the `jx` console output. It uses CloudFormation to set up EKS as well as worker nodes, so you can monitor the progress by visiting the CloudFormation page.

Jenkins X and other installations

Next, the installation of Jenkins X itself and a few other applications (e.g., ChartMuseum for storing Helm charts) will start. The exact list of apps that will be installed depends on the Kubernetes flavor, the type of setup, and the hosting vendor. But, before it proceeds, it will ask us a few other questions; What kind do we want to install? Static or serverless? Please answer with `Serverless Jenkins X Pipelines with Tekton`. Even though Jenkins X started its history with Jenkins, the preferred pipeline engine is Tekton, which is available through the serverless flavor of Jenkins X. We'll discuss Tekton and the reasons Jenkins X is using it later.

Ingress installation

The next in line is Ingress. The process will try to find it inside the `kube-system` Namespace and install it if it's not there. The process installs it through a Helm chart. As a result, Ingress will create a load balancer that will provide an entry point into the cluster.

Custom domain name

Jenkins X recommends using a custom DNS name to access services in your Kubernetes cluster. However, there is no way for me to know if you have a domain available for use or not. Instead, we'll use the nip.io service to create a fully qualified domain. To do that, we'll have to answer with `n` to the question `"would you like to register a wildcard DNS ALIAS to point at this ELB address?"`. As a result, we'll be presented with another question. `"Would you like to wait and resolve this address to an IP address and use it for the domain?"`. Answer with `y` or press the enter key since that is the default answer. The process will wait until the Elastic Load Balancer (ELB) is created and use its hostname to deduce its IP.

Long-term logs

You might be asked *to enable long-term logs storage*. Make sure to answer with `n`.

We will not need it for our exercises and, at the time of this writing, it is still in the

"experimental" phase.

Git and Github settings

Next, we'll be asked a few questions relating to Git and GitHub. You should be able to answer those. In most cases, all you have to do is confirm the suggested answer by pressing the enter key. As a result, `jx` will store the credentials internally so that it can continue interacting with GitHub on our behalf. It will also install the software necessary for those environments (Namespaces) to function correctly inside our cluster.

Organization

We're almost done. Only one question is pending; `Select the organization where you want to create the environment repository?` Choose one from the list.

GitOps

The process will create two GitHub repositories; `environment-jx-rocks-staging` that describes the staging environment and `environment-jx-rocks-production` for production. Those repositories will hold the definitions of those environments. For example, when you decide to promote a release to production, your pipelines will not install anything directly. Instead, they will push changes to `environment-jx-rocks-production` which, in turn, will trigger another job that will comply with the updated definition of the environment.

That's GitOps.

Nothing is done without recording a change in Git. Of course, for that process to work, we need new jobs in Jenkins X, so the process created two jobs that correspond to those repositories. We'll discuss the environments in greater detail later.

`kubectl` context change

Finally, the `kubectl` context was changed to point to the `jx` Namespace, instead of `default`.

As you can see, a single `jx create cluster` command did a lot of heavy lifting. Nevertheless, there is one piece missing; it did not create a Cluster Autoscaler as it's

not currently part of EKS. We'll add it ourselves so that we don't need to worry about whether the cluster needs more nodes.

Creating a Cluster Autoscaler

We'll add a few tags to the Autoscaling Group dedicated to worker nodes. To do that, we need to discover the name of the group. Fortunately, names follow a pattern which we can use to filter the results.

Retrieving matching autoscaling groups

First, we'll retrieve the list of the AWS Autoscaling Groups, and filter the result with `jq` so that only the name of the matching group is returned.

```
ASG_NAME=$(aws autoscaling \
    describe-auto-scaling-groups \
    | jq -r ".AutoScalingGroups[] \
    | select(.AutoScalingGroupName \
    | startswith(\"eksctl-jx-rocks-nodegroup\")) \
    .AutoScalingGroupName")

echo $ASG_NAME
```

We retrieved the list of all the groups and filtered the output with `jq` so that only those with names that start with `eksctl-$NAME-nodegroup` are returned. Finally, that same `jq` command retrieved the `AutoScalingGroupName` field and we stored it in the environment variable `ASG_NAME`. The last command output the group name so that we can confirm (visually) that it looks correct.

Adding tags to the group

Next, we'll add a few tags to the group. Kubernetes Cluster Autoscaler will work with the one that has the `k8s.io/cluster-autoscaler/enabled` and `kubernetes.io/cluster/[NAME_OF_THE_CLUSTER]` tags. So, we just have to add those tags to let Kubernetes know which group to use.

```
aws autoscaling \
    create-or-update-tags \
    --tags \
    ResourceId=$ASG_NAME,ResourceType=auto-scaling-group,Key=k8s.io/cluster-autoscaler/enabled,Valu
    ResourceId=$ASG_NAME,ResourceType=auto-scaling-group,Key=kubernetes.io/cluster/jx-rocks,Value=
```

Retrieve the name of the role

The last change we'll have to do in AWS is add a few additional permissions to the role. Just as with the Autoscaling Group, we do not know the name of the role, but

we know the pattern used to create it. Therefore, we'll retrieve the name of the role before we add a new policy to it.

```
IAM_ROLE=$(aws iam list-roles \
    | jq -r ".Roles[] \
    | select(.RoleName \
    | startswith(\"eksctl-jx-rocks-nodegroup\")) \
    .RoleName")

echo $IAM_ROLE
```

We listed all the roles and we used `jq` to filter the output so that only the one with the name that starts with `eksctl-jx-rocks-nodegroup-0-NodeInstanceRole` is returned. Once we filtered the roles, we retrieved the `RoleName` and stored it in the environment variable `IAM_ROLE`.

Define and add the new policy to the role

Next, we need JSON that describes the new policy that will allow a few additional actions related to `autoscaling`. I already prepared one and stored it in the vfarcic/k8s-specs repository.

Now, let's `put` the new policy to the role.

```
aws iam put-role-policy \
    --role-name $IAM_ROLE \
    --policy-name jx-rocks-AutoScaling \
    --policy-document https://raw.githubusercontent.com/vfarcic/k8s-specs/master/scaling/eks-autos
```

Installing `cluster-autoscaler`

Now that we have added the required tags to the Autoscaling Group and created the additional permissions that will allow Kubernetes to interact with the group, we can install the *cluster-autoscaler* Helm Chart from the stable channel. All we have to do now is execute `helm install stable/cluster-autoscaler`. However, since tiller (server-side Helm) has a lot of problems, Jenkins X does not use it by default. So, instead of using `helm install` command, we'll run `helm template` to output YAML files that we can use with `kubectl apply`.

```
mkdir -p charts

helm fetch stable/cluster-autoscaler \
    -d charts \
    --untar

mkdir -p k8s-specs/aws
```

```
helm template charts/cluster-autoscaler \
    --name aws-cluster-autoscaler \
    --output-dir k8s-specs/aws \

    --namespace kube-system \
    --set autoDiscovery.clusterName=jx-rocks \
    --set awsRegion=$AWS_DEFAULT_REGION \
    --set sslCertPath=/etc/kubernetes/pki/ca.crt \
    --set rbac.create=true

kubectl apply \
    -n kube-system \
    -f k8s-specs/aws/cluster-autoscaler/*
```

Once the Deployment is rolled out, the autoscaler should be fully operational.

You can see from the Cluster Autoscaler (CA) example how much `jx` helps. It took a single `jx` command to:

- create a cluster
- configure a cluster
- install a bunch of tools into that cluster
- and so on

For the only thing that `jx` did not do (creating the Cluster Autoscaler), we only had to execute five or six commands, not counting the effort I had to put in to figuring them out. Hopefully, EKS CA will be part of `jx` soon.

We'll get back to the new cluster and the tools that were installed and configured in the What Did We Get? section. Feel free to jump there if you have no interest in other Cloud providers or how to install Jenkins X inside an existing cluster.

Next up: AKS.