Configure an EKS cluster in AWS

EKS is Amazon’s managed Kubernetes service.

Fundamentally, a new cluster can be configured through the AWS UI and/or the `aws` CLI tool.
There are a number of other tools that can make use of the `aws` CLI tool, however, such as `eksctl` and `terraform`, which yield benefits such as being able to store your configuration in files.

1. Deploy an EKS cluster

Prerequisites

- The Access Key and Secret of an AWS account with permission to create resources and IAM roles.
- `aws` CLI tool, configured by running `aws configure`, the above details added, and a default region defined.
- `eksctl` installed

On a Mac, it’s as simple as:

```
brew tap weaveworks/tap
brew install weaveworks/tap/eksctl
```

Windows users can use chocolatey:

```
chocolatey install eksctl
```

https://eksctl.io/introduction/installation/

Deployment

In principle, setting up an EKS cluster requires nothing more than running:

```
eksctl create cluster
```

This command will create cluster with the default parameters:

- an auto-generated name, e.g. `fabulous-mushroom-1527688624` - highly creative list, beware! 😀
- 2x m5.large nodes (this instance type suits most common use-cases, and is good value for money)
- uses official AWS EKS AMI
- in the default region defined in your AWS CLI configuration
- dedicated VPC (check your quotas)
- using static AMI resolver

However, there are a number of additional options. Use the option for adding in a **config.yaml** file in order to specify at least 3 replicas for the deploy:

```
eksctl create cluster \
#pass in a yaml file with pre-defined values (handy for repeatability
and auditing)
--config-file=<path> \
#automatically configure your kubeconfig file (doesn't seem to actually
work)
--auto-kubeconfig \
#define key to use to access the nodes via SSH
--ssh-access --ssh-public-key=~/.ssh/gospel_eks.pem \
#define the boot volumes for the cluster nodes
--node-volume-size=50 --node-volume-type=io1
```

Use this example for the **config.yaml** file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: gospel-test-2
  region: eu-west-1

nodeGroups:
  - name: ng-2
    instanceType: m5.xlarge
    desiredCapacity: 3
    privateNetworking: true
```

CloudWatch logging will not be enabled for your cluster by default, but you can enable it with:

```
eksctl utils update-cluster-logging --region=<aws region>
--cluster=<cluster name> --enable-types all --approve
```

Troubleshooting

If you encounter problems creating your Kubernetes nodes, try this document:

https://docs.aws.amazon.com/eks/latest/userguide/worker_node_IAM_role.html

Cleanup

In case you need to clean up resources, run the following command:

```
eksctl delete cluster --region=eu-west-1 --name=gospel-test-cluster-1
```

2. Deploy Kubernetes Dashboard

The Kubernetes dashboard offers a web UI for managing Kubernetes resources. In order to set it up follow these steps:

1. Install metrics-server from GitHub on an Amazon EKS cluster using `curl` and `jq`

Open a terminal window and navigate to a directory where you would like to download the latest metrics-server release.

Copy and paste the commands below into your terminal window and type Enter to execute them. These commands download the latest release, extract it, and apply the version 1.8+ manifests to your cluster.

```
DOWNLOAD_URL=$(curl --silent
"https://api.github.com/repos/kubernetes-sigs/metrics-server/releases/la
test" | jq -r .tarball_url)
DOWNLOAD_VERSION=$(grep -o '[^/v]*$' <<< $DOWNLOAD_URL)
curl -Ls $DOWNLOAD_URL -o metrics-server-$DOWNLOAD_VERSION.tar.gz
mkdir metrics-server-$DOWNLOAD_VERSION
tar -xzf metrics-server-$DOWNLOAD_VERSION.tar.gz --directory
metrics-server-$DOWNLOAD_VERSION --strip-components 1
kubectl apply -f metrics-server-$DOWNLOAD_VERSION/deploy/1.8+/
```

Verify that the metrics-server deployment is running the desired number of pods with the following command.

```
kubectl get deployment metrics-server -n kube-system
```

Output:

```
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
metrics-server    1         1         1            1           56m
```

2. Deploy the Dashboard

Use the following command to deploy the Kubernetes dashboard.

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta6/aio/
deploy/recommended.yaml
```

Output:

```
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard
created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

3. Create an eks-admin Service Account and Cluster Role Binding

By default, the Kubernetes dashboard user has limited permissions. In this section, you create an eks-admin service account and cluster role binding that you can use to securely connect to the dashboard with admin-level permissions. For more information, see Managing Service Accounts in the Kubernetes documentation.

To create the eks-admin service account and cluster role binding

Important

The example service account created with this procedure has full cluster-admin (superuser) privileges on the cluster. For more information, see Using RBAC Authorization in the Kubernetes documentation.

Create a file called **eks-admin-service-account.yaml** with the text below. This manifest defines a service account and cluster role binding called eks-admin.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: eks-admin
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: eks-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: eks-admin
  namespace: kube-system
```

Apply the service account and cluster role binding to your cluster.

```
kubectl apply -f eks-admin-service-account.yaml
```

Output:

```
serviceaccount "eks-admin" created
clusterrolebinding.rbac.authorization.k8s.io "eks-admin" created
```

4. Connect to the Dashboard

Now that the Kubernetes dashboard is deployed to your cluster, and you have an administrator service account that you can use to view and control your cluster, you can connect to the dashboard with that service account.

To connect to the Kubernetes dashboard

Retrieve an authentication token for the eks-admin service account. Copy the <authentication_token> value from the output. You use this token to connect to the dashboard.

```
kubectl -n kube-system describe secret $(kubectl -n kube-system get
secret | grep eks-admin | awk '{print $1}')
```

Output:

```
Name:         eks-admin-token-b5zv4
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name=eks-admin


kubernetes.io/service-account.uid=bcfe66ac-39be-11e8-97e8-026dce96b6e8
Type:  kubernetes.io/service-account-token
Data
====
ca.crt:     1025 bytes
namespace:  11 bytes
token:      <authentication_token>
```

Start the kubectl proxy:

```
kubectl proxy
```

To access the dashboard endpoint, open the following link with a web browser: http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#!/login

Choose Token, paste the <authentication_token> output from the previous command into the Token field, and choose SIGN IN.

Note It may take a few minutes before CPU and memory metrics appear in the dashboard.

5. Next Steps

After you have connected to your Kubernetes cluster dashboard, you can view and control your cluster using your **eks-admin** service account. For more information about using the dashboard, see the project documentation on GitHub.

3. Set up an EFS file share

In order to create persistent storage for the EKS cluster you will need to create an EFS volume and associate it with your deployment. EFS file shares can be set up from the CLI, but the UI is relatively straight forward too.

Steps to provision it from the AWS Console:

1. In the AWS console look for **EFS** - select **Create file system**
2. Select the same **VPC** you used when creating your EKS cluster.
3. Select all the Availability Zones in which you have EKS nodes.
4. In the AWS console go to **EC2** - **Network &Security** - **Security Groups**
5. Create a security group in EC2 allowing access to the NFS service (TCP 2049) with the security group assigned to your EKS nodes as the “Source”. Your EKS nodes' security group might be called something like `eksctl-gospel-test-nodegroup-ng-2-SG-XXXXXXX` XXXXXX.

N.B. Editing your EKS nodegroup’s security group means that it now has dependencies that eksctl doesn’t know about/ doesn’t have permission to change. If you eventually decide to delete your cluster using `eksctl delete cluster`, it will fail unless you either revert these changes or manually delete the security group.

4. Configure Kubernetes to access EFS file share

Create the namespace in which you’d like to deploy Gospel:

```
kubectl create namespace gospel
```

Create a `storageclass.yaml` file with a definition of your eventual storageClass's **name** and **namespace**:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gospel-efs
  namespace: gospel
provisioner: example.com/aws-efs
```

In a `configmap.yaml` file, change **file.system.id** and **aws.region** to the values appropriate for your EFS:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: efs-provisioner
  namespace: gospel
data:
  file.system.id: fs-1234abcd
  aws.region: eu-west-1
  provisioner.name: example.com/aws-efs
  dns.name: ""
```

This configMap will be used later for environment variables in a pod that will allow access from your namespace to EFS.
Note: (Optional) To mount the EFS by your own DNS name instead of the DNS name assigned by AWS (file-system-id.efs.aws-region.amazonaws.com), use the **dns.name** property.

To apply the storageClass and configMap to your cluster, run the following commands:

```
kubectl apply -f storageclass.yaml
kubectl apply -f configmap.yaml
```

Create an `rbac.yaml`, define the permissions that are needed to create Kubernetes resources and run correctly:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: efs-provisioner-runner
rules:
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["list", "watch", "create", "update", "patch"]
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: run-efs-provisioner
subjects:
  - kind: ServiceAccount
    name: efs-provisioner
     # replace with namespace where provisioner is deployed
    namespace: gospel
roleRef:
  kind: ClusterRole
  name: efs-provisioner-runner
  apiGroup: rbac.authorization.k8s.io
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-efs-provisioner
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-efs-provisioner
subjects:
  - kind: ServiceAccount
    name: efs-provisioner
    # replace with namespace where provisioner is deployed
    namespace: gospel
roleRef:
    kind: Role
    name: leader-locking-efs-provisioner
    apiGroup: rbac.authorization.k8s.io
```

Note: There are versions of this file online with fewer permissions in the `ClusterRole` definition- they do not allow access to `endpoints` and less access to `events`. Your pvc will not mount if these permissions are not present.

In a `deployment.yaml` file, change the **volumes** configuration to a **path** of ***/***. Make sure the **nfs** configuration references the **file.system.id** you referenced in `configmap.yaml`. See the following example:

```
kind: ServiceAccount
apiVersion: v1
metadata:
  name: efs-provisioner
  namespace: gospel
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: efs-provisioner
  namespace: gospel
spec:
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: efs-provisioner
    spec:
      serviceAccount: efs-provisioner
      containers:
        - name: efs-provisioner
          image: quay.io/external_storage/efs-provisioner:latest
          env:
            - name: FILE_SYSTEM_ID
              valueFrom:
                configMapKeyRef:
                  name: efs-provisioner
                  key: file.system.id
            - name: AWS_REGION
              valueFrom:
                configMapKeyRef:
                  name: efs-provisioner
                  key: aws.region
            - name: DNS_NAME
              valueFrom:
                configMapKeyRef:
                  name: efs-provisioner
              key: dns.name
              optional: true
        - name: PROVISIONER_NAME
          valueFrom:
            configMapKeyRef:
              name: efs-provisioner
              key: provisioner.name
      volumeMounts:
        - name: pv-volume
          mountPath: /persistentvolumes
  volumes:
    - name: pv-volume
```

```
            nfs:
              server: fs-1234abcd.efs.eu-west-1.amazonaws.com
              path: /
```

You can see the env vars in this file that were defined in the configMap we just created.

Note: This updated configuration allows the efs-provisioner to create child directories to back each persistent volume that it provisions at the root of the EFS volume.

To apply the RBAC resources and the Deployment manifest:

```
kubectl apply -f rbac.yaml
kubectl apply -f deployment.yaml
```

Once you have created your deployment, you can test your access by connecting to your `efs-provisioner` pod and checking if you can create a file in `/persistentvolumes`.

```
$ kubectl exec -it -n gospel efs-provisioner-1234a567bc-defgh sh
/ # ls /persistentvolumes/
/ # mkdir /persistentvolumes/test
/ # ls /persistentvolumes/
test
```

5. Provision EBS Volumes

To deploy the Amazon EBS CSI Driver to an Amazon EKS cluster

You can also use the steps listed in AWS CSI driver official documentation.

1. Create an IAM policy called `Amazon_EBS_CSI_Driver` for your worker node instance profile that allows the Amazon EBS CSI Driver to make calls to AWS APIs on your behalf. Use the following AWS CLI commands to create the IAM policy in your AWS account. You can view the policy document on GitHub.
    a. Download the policy document from GitHub.

    ```
    curl -O
    https://raw.githubusercontent.com/kubernetes-sigs/aws-ebs-csi-d
    river/v0.4.0/docs/example-iam-policy.json
    ```

    b. Create the policy.

    ```
    aws iam create-policy --policy-name Amazon_EBS_CSI_Driver \
    --policy-document file://example-iam-policy.json
    ```

    Take note of the policy ARN that is returned.
2. Get the IAM role name for your worker nodes. Use the following command to print the `aws-auth` configmap.

```
kubectl -n kube-system describe configmap aws-auth
```

Output:

```
Name:         aws-auth
Namespace:    kube-system
Labels:       <none>
Annotations:  <none>

Data
====
mapRoles:
----
- groups:
  - system:bootstrappers
  - system:nodes
  rolearn:
arn:aws:iam::111122223333:role/eksctl-alb-nodegroup-ng-b1f603c5-Nod
eInstanceRole-GKNS581EASPU
  username: system:node:{{EC2PrivateDNSName}}

Events:  <none>
```

Record the role name for any `rolearn` values that have the `system:nodes` group assigned to them. In the previous example output, the role name is `eksctl-alb-nodegroup-ng-b1f603c5-NodeInstanceRole-GKNS581EASPU`. You should have one value for each node group in your cluster.

3. Attach the new `Amazon_EBS_CSI_Driver` IAM policy to each of the worker node IAM roles you identified earlier with the following command, substituting the red text with your own AWS account number and worker node IAM role name.

```
aws iam attach-role-policy \
--policy-arn arn:aws:iam::111122223333:policy/Amazon_EBS_CSI_Driver
\
--role-name
eksctl-alb-nodegroup-ng-b1f603c5-NodeInstanceRole-GKNS581EASPU
```

4. Deploy the Amazon EBS CSI Driver with the following command.

Note

This command requires version 1.14 or greater of `kubectl`. You can see your `kubectl` version with the following command. To install or upgrade your `kubectl` version, see Installing kubectl.

```
kubectl version --client --short
kubectl apply -k
"github.com/kubernetes-sigs/aws-ebs-csi-driver/deploy/kubernetes/ov
erlays/stable/?ref=master"
```

To deploy a sample application and verify that the CSI driver is working

This procedure uses the Dynamic Volume Provisioning example from the Amazon EBS Container Storage Interface (CSI) Driver GitHub repository to consume a dynamically-provisioned Amazon EBS volume.

1. Clone the Amazon EBS Container Storage Interface (CSI) Driver GitHub repository to your local system.

   ```
   git clone https://github.com/kubernetes-sigs/aws-ebs-csi-driver.git
   ```

2. Navigate to the `dynamic-provisioning` example directory.

   ```
   cd aws-ebs-csi-driver/examples/kubernetes/dynamic-provisioning/
   ```

3. Deploy the `ebs-sc` storage class, `ebs-claim` persistent volume claim, and `app` sample application from the `specs` directory.

   ```
   kubectl apply -f specs/
   ```

4. Describe the `ebs-sc` storage class.

   ```
   kubectl describe storageclass ebs-sc
   ```

   Output:

```
Name:            ebs-sc
IsDefaultClass:  No
Annotations:
kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"sto
rage.k8s.io/v1","kind":"StorageClass","metadata":{"annotations":{},
"name":"ebs-sc"},"provisioner":"ebs.csi.aws.com","volumeBindingMode
":"WaitForFirstConsumer"}

Provisioner:           ebs.csi.aws.com
Parameters:            <none>
AllowVolumeExpansion:  <unset>
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

Note that the storage class uses the `WaitForFirstConsumer` volume binding mode. This means that volumes are not dynamically provisioned until a pod makes a persistent volume claim. For more information, see Volume Binding Mode in the Kubernetes documentation.

5. Watch the pods in the default namespace and wait for the `app` pod to become ready.

```
kubectl get pods --watch
```

6. List the persistent volumes in the default namespace. Look for a persistent volume with the `default/ebs-claim` claim.

```
kubectl get pv
```

Output:

```
NAME                                       CAPACITY   ACCESS MODES
RECLAIM POLICY   STATUS   CLAIM               STORAGECLASS   REASON
AGE
pvc-37717cd6-d0dc-11e9-b17f-06fad4858a5a   4Gi        RWO
Delete           Bound    default/ebs-claim   ebs-sc
30s
```

7. Describe the persistent volume.

```
kubectl describe pv pvc-37717cd6-d0dc-11e9-b17f-06fad4858a5a
```

Output:

```
Name:              pvc-37717cd6-d0dc-11e9-b17f-06fad4858a5a
Labels:            <none>
Annotations:       pv.kubernetes.io/provisioned-by: ebs.csi.aws.com
Finalizers:        [kubernetes.io/pv-protection
external-attacher/ebs-csi-aws-com]
StorageClass:      ebs-sc
Status:            Bound
Claim:             default/ebs-claim
Reclaim Policy:    Delete
Access Modes:      RWO
VolumeMode:        Filesystem
Capacity:          4Gi
Node Affinity:
  Required Terms:
    Term 0:        topology.ebs.csi.aws.com/zone in [us-west-2a]
Message:
Source:
    Type:              CSI (a Container Storage Interface (CSI)
volume source)
    Driver:            ebs.csi.aws.com
    VolumeHandle:      vol-0d651e157c6d93445
    ReadOnly:          false
    VolumeAttributes:
storage.kubernetes.io/csiProvisionerIdentity=1567792483192-8081-ebs
.csi.aws.com
Events:                <none>
```

The Amazon EBS volume ID is listed as the `VolumeHandle`.

8. Verify that the pod is successfully writing data to the volume.

```
kubectl exec -it app cat /data/out.txt
```

Output:

```
Fri Sep 6 19:26:53 UTC 2019
Fri Sep 6 19:26:58 UTC 2019
Fri Sep 6 19:27:03 UTC 2019
Fri Sep 6 19:27:08 UTC 2019
Fri Sep 6 19:27:13 UTC 2019
Fri Sep 6 19:27:18 UTC 2019
```

9. When you finish experimenting, delete the resources for this sample application to clean up.

```
kubectl delete -f specs/
```

6. Configure Kubernetes deployment files to use AWS Storage classes

Define a pvc that will claim the storageClass we created at the beginning:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: gospel-test-data
  namespace: gospel
  annotations:
    volume.beta.kubernetes.io/storage-class: "gospel-efs"
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
```

Finally mount that pvc in your podDeployment

```
volumes:
      - name: your-pods-data
        persistentVolumeClaim:
            claimName: gospel-test-data
```

Further Reading:

EKS:

https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html

https://eksctl.io/introduction/getting-started/

https://docs.aws.amazon.com/eks/latest/userguide/dashboard-tutorial.html

EBS:

https://docs.aws.amazon.com/eks/latest/userguide/ebs-csi.html

EFS:

https://aws.amazon.com/premiumsupport/knowledge-center/eks-pods-efs/

https://docs.aws.amazon.com/eks/latest/userguide/efs-csi.html

https://docs.aws.amazon.com/efs/latest/ug/accessing-fs-create-security-groups.html

Next Steps:

Continue to deploy Gospel using the Gospel Deployment Guide - all clouds