Introduction to Crossplane

Pavan Kumar
Feb 27 · 6 min read

How to create any resource on the cloud using Kubernetes manifests and Crossplane.

In the Kubernetes era, all of your application blueprints are packaged into a lot of Kubernetes manifests files or maybe also packages as charts using tools like helm. So how do you create any cloud resource on the cloud? You can maybe use

1. An external terraform module to create the resource.
2. Use a Kubernetes Job and create the resources using AWS SDK's.
3. Use a bash / Python script and internally call AWS CLI commands.

But how reliable is this? Unlike Kubernetes manifests in which the yaml file can be edited on the fly, every time an attribute changes you will have to explicitly call these dependent resources. And in the modern GitOps era having such external dependencies might not be a feasible option for your GitOps Solutions. How do we fix this then? Here comes **Crossplane** into the picture. Crossplane enables you to provision, compose, and consume infrastructure in any **cloud service provider using the Kubernetes API**. Using Crossplane you can create resources on the cloud using simple manifests and can then integrate this with your CI/CD or GitOps pipelines. Crossplane is an open-source project. It is started by Upbound and then later got adopted by the CNCF as a sandbox project.

Image Credits: Crossplane

What is the entire story all about? (TLDR)

1. Install Cross plane on our Kubernetes Cluster (AKS, GKE, EKS, KIND)
2. Configure Crossplane to communicate with AWS.
3. Install required packages for CrossPlane to communicate with AWS.
4. Create a VPC, SG, RDS using CrossPlane from our Kubernetes Cluster.
5. Verify that the resources have been created from the AWS Console.

Prerequisites

1. A Kubernetes cluster (Can be either On-Prem, AKS, EKS, GKE, Kind).
2. An AWS account.

Story Resources

1. **GitHub Link**: https://github.com/pavan-kumar-99/medium-manifests
2. **GitHub Branch**: crossplane

Install Cross plane in a Kubernetes Cluster

You can use an existing Kubernetes cluster for this demo. Alternatively, you can also install a Kubernetes cluster using kind or using GitHub actions. You can refer to my previous articles on how to create a Kubernetes cluster using

1. **GitHub actions**
2. **Kind**

Resources

1. GitHub Link: https://github.com/pavan-kumar-99/medium-manifests/tree/crossplane

Once you have the Kubernetes cluster created let us now install Crossplane in our cluster. You can clone my repo with the crossplane branch for all the manifests used in this article.

```
###Clone the repo
```

```
git clone https://github.com/pavan-kumar-99/medium-manifests.git -b crossplane

cd medium-manifests/crossplane-aws

#Create the namespace and install the components using helm

kubectl create namespace crossplane-system

helm repo add crossplane-stable https://charts.crossplane.io/stable
helm repo update

helm install crossplane --namespace crossplane-system crossplane-stable/crossplane

#Check the components are up and healthy

kubectl get all -n crossplane-system
```

(OR)

```
git clone https://github.com/pavan-kumar-99/medium-manifests.git -b crossplane

cd medium-manifests/crossplane-aws

make install_crossplane
```

Alternatively, you can also use a makefile that I have written. This will install kind in your MAC / Linux machines, create a Kind cluster and then install crossplane in the Kind cluster.

```
.PHONY : all install_kind_linux install_kind_mac create_kind_cluster

KIND_VERSION := $(shell kind --version 2>/dev/null)

install_kind_linux :
ifdef KIND_VERSION
        @echo "Found version $(KIND_VERSION)"
else
        @curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.10.0/kind-linux-amd64
        @chmod +x ./kind
        @mv ./kind /bin/kind
endif

install_kind_mac :
ifdef KIND_VERSION
        @echo "Found version $(KIND_VERSION)"
else
```

```
18          @brew install kind
19  endif
20
21  create_kind_cluster :
22          @kind create cluster --name crossplane-cluster
23          @kind get kubeconfig --name crossplane-cluster
24          @kubectl config set-context kind-crossplane-cluster
25
26  install_crossplane :
27          @kubectl create namespace crossplane-system
28          @helm repo add crossplane-stable https://charts.crossplane.io/stable
29          @helm repo update
30          @helm install crossplane --namespace crossplane-system crossplane-stable/crossplane
31
32  all : install_kind_linux create_kind_cluster create_kind_cluster
```

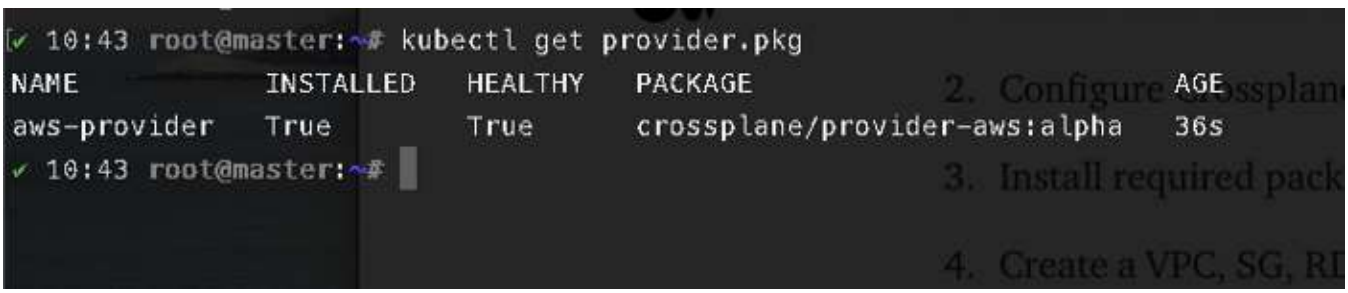
Makefile hosted with ♡ by GitHub view raw

Let us now install the AWS Provider. This will Install all the CRD's (Custom Resources Definitions) required to create resources on the cloud. Ex: **rdsinstances.database.aws.crossplane.io**, **ec2.aws.crossplane.io/v1alpha1**, etc.

```
kubectl apply -f aws-provider.yaml
```

```
1  apiVersion: pkg.crossplane.io/v1
2  kind: Provider
3  metadata:
4    name: aws-provider
5  spec:
6    package: crossplane/provider-aws:alpha
```

aws-provider.yaml hosted with ♡ by GitHub view raw



Provider Package

###Once you install the Provider, wait for the Provider to be healthy by executing

```
kubectl get provider.pkg
```

Once the Provider is healthy let us now configure the Provider to communicate with AWS by creating a `ProviderConfig` definition. Make sure that you have already configured your credentials using **AWS configure (From the cli, if you are running the commands from a local cluster).**

###Generate the configuration files with the AWS Credentials.

```
AWS_PROFILE=default && echo -e "[default]\naws_access_key_id = $(aws configure get aws_access_key_id --profile $AWS_PROFILE)\naws_secret_access_key = $(aws configure get aws_secret_access_key --profile $AWS_PROFILE)" > creds.conf
```

###Create a Kubernetes secret with the configuration file generated.

```
kubectl create secret generic aws-secret-creds -n crossplane-system --from-file=creds=./creds.conf
```

###Once the secret is created let us now create the Provider config for our AWS account.

```
kubectl apply -f provider-config.yaml
```

```yaml
apiVersion: aws.crossplane.io/v1beta1
kind: ProviderConfig
metadata:
  name: awsconfig
spec:
  credentials:
    source: Secret
    secretRef:
      namespace: crossplane-system
      name: aws-secret-creds
      key: creds
```

provider-config.yaml hosted with ♡ by GitHub view raw

Upon successful creation, your local cluster should now be able to communicate with AWS. Let us now try creating the following scenario. Let us create a VPC and a Security

Group that would allow access from Port 3306 from anywhere from the world. Let us simultaneously create an RDS and attach the aforementioned SG to the same RDS Instance so that it would be publically accessible. Once this resource is created we will create a pod in our local cluster and check if it can access the RDS Instance. Seems good? Let us now get into action.

Let us create a VPC in the us-east-1 region with the below-mentioned spec.

```yaml
apiVersion: ec2.aws.crossplane.io/v1beta1
kind: VPC
metadata:
  name: production-vpc
spec:
  forProvider:
    region: us-east-1
    cidrBlock: 192.168.0.0/16
    enableDnsSupport: true
    enableDnsHostNames: true
    tags:
    - key: Environment
      value: Production
    - key: Owner
      value: Pavan
    - key: Name
      value: production-vpc
    instanceTenancy: default
  providerConfigRef:
    name: awsconfig
```

aws-vpc.yaml hosted with ♡ by GitHub view raw

```
kubectl apply -f aws-vpc.yaml
```

###Let us check the status of the VPC. We are now referring to the provider created earlier in (line no 20).

```
kubectl get vpc
```

VPC created with VPC ID and CIDR block and the sync and Ready state

Once our VPC is successfully created let us create 2 subnets and attach an internet gateway to our VPC and also add a Route table for the same so that we can create our RDS in these Public subnets and then access from our local pod. However this is not the suggested method in Production, you should never spin your RDS in a Public Subnet in a Production environment.

```yaml
apiVersion: ec2.aws.crossplane.io/v1beta1
kind: Subnet
metadata:
  name: prod-subnet-1
spec:
  forProvider:
    region: us-east-1
    availabilityZone: us-east-1a
    cidrBlock: 192.168.1.0/24
    vpcIdRef:
      name: production-vpc
    tags:
    - key: Environment
      value: Production
    - key: Name
      value: prod-subnet-1
    - key: Owner
      value: Pavan
    mapPublicIPOnLaunch: true
  providerConfigRef:
    name: awsconfig

---

apiVersion: ec2.aws.crossplane.io/v1beta1
kind: Subnet
metadata:
  name: prod-subnet-2
spec:
  forProvider:
    region: us-east-1
    availabilityZone: us-east-1b
    cidrBlock: 192.168.2.0/24
    vpcIdRef:
      name: production-vpc
    tags:
    - key: Environment
      value: Production
```

```
        value: Production
      - key: Name
        value: prod-subnet-2
      - key: Owner
        value: Pavan
      mapPublicIPOnLaunch: true
    providerConfigRef:
      name: awsconfig
```

aws-subnet.yaml hosted with ♡ by GitHub view raw

```
kubectl apply -f aws-subnet.yaml
```

###Let us check the status of the subnets.

```
kubectl get subnets
```

Let us now create the corresponding Internet gateway and Route table.

```
apiVersion: ec2.aws.crossplane.io/v1beta1
kind: InternetGateway
metadata:
  name: production-internetgateway
spec:
  forProvider:
    region: us-east-1
    vpcIdRef:
      name: production-vpc
  providerConfigRef:
    name: awsconfig

---


apiVersion: ec2.aws.crossplane.io/v1alpha4
kind: RouteTable
metadata:
  name: production-routetable
spec:
  forProvider:
    region: us-east-1
    routes:
      - destinationCidrBlock: 0.0.0.0/0
        gatewayIdRef:
          name: production-internetgateway
    associations:
```

```
      - subnetIdRef:
          name: prod-subnet-2
      - subnetIdRef:
          name: prod-subnet-1
    vpcIdRef:
      name: production-vpc
  providerConfigRef:
    name: awsconfig
```

aws-igwrt.yaml hosted with ♡ by GitHub view raw

```
kubectl apply -f aws-igwrt.yaml
```

###Let us check the status of the Route table and Internet Gateway

```
kubectl get InternetGateway,RouteTable
```

Let us now create the security group that would allow communication over port 3306 to the Internet. Later we will attach this security group to our RDS Instance.

```
apiVersion: ec2.aws.crossplane.io/v1beta1
kind: SecurityGroup
metadata:
  name: rds-access-sg
spec:
  forProvider:
    region: us-east-1
    vpcIdRef:
      name: production-vpc
    groupName: mysql-sg
    description: RDS communication to local application Pods
    tags:
    - key: Environment
      value: Production
    - key: Owner
      value: Pavan
    - key: Name
      value: rds-access-sg
    ingress:
      - fromPort: 3306
        toPort: 3306
        ipProtocol: tcp
        ipRanges:
          - cidrIp: 0.0.0.0/0
  providerConfigRef:
```

```
    name: awsconfig
```

aws-sg.yaml hosted with ♡ by GitHub view raw

```
kubectl apply -f aws-sg.yaml
```

###Let us check the status of the Route table and Internet Gateway

```
kubectl get SecurityGroup
```

Security group created in the afore-mentioned VPC.

Let us now create the RDS instance, but before we do that we would need a subnet group in which the RDS instance has to be created. We will use the subnets created earlier in the DB Subnet Group.

```
apiVersion: database.aws.crossplane.io/v1beta1
kind: DBSubnetGroup
metadata:
  name: prod-subnet-group
spec:
  forProvider:
    region: us-east-1
    description: "Prod Subnet group"
    subnetIdRefs:
      - name: prod-subnet-2
      - name: prod-subnet-1
  providerConfigRef:
    name: awsconfig

---

apiVersion: database.aws.crossplane.io/v1beta1
kind: RDSInstance
metadata:
  name: production-rds
spec:
  forProvider:
    allocatedStorage: 50
    autoMinorVersionUpgrade: true
    applyModificationsImmediately: false
```

```
    backupRetentionPeriod: 0
    caCertificateIdentifier: rds-ca-2019
    copyTagsToSnapshot: false
    dbInstanceClass: db.t2.small
    dbSubnetGroupName: prod-subnet-group
    vpcSecurityGroupIDRefs:
      name: rds-access-sg
    deletionProtection: false
    enableIAMDatabaseAuthentication: false
    enablePerformanceInsights: false
    engine: mysql
    region: us-east-1
    engineVersion: 5.6.35
    finalDBSnapshotIdentifier: muvaf-test
    licenseModel: general-public-license
    masterUsername: admin
    multiAZ: true
    port: 3306
    tags:
    - key: Name
      value: production-rds
    - key: Environment
      value: Production
    - key: Owner
      value: Pavan
    publiclyAccessible: true
    storageEncrypted: false
    storageType: gp2
  providerConfigRef:
    name: awsconfig
  writeConnectionSecretToRef:
    name: production-rds-conn-string
    namespace: default
```

kubectl apply -f aws-rds.yaml

###Let us check the status of the RDS Instance. The credentials are stored in a secret called production-rds-conn-string in the default namespace. (line no 56)

kubectl get RDSInstance

The AWS Infra is now ready to serve us.....

Now let us try to access our Mysql RDS Instance. We can access this by decoding the secret **production-rds-conn-string** created in the default namespace. You can connect to the database using the MySQL client

> *mysql -h <hostname> -u <user_name> -p <password>*

Alternatively, you can spin up a pod and connect from the pod itself.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql-rds-connection-test
  namespace: default
spec:
  containers:
  - name: mysqlconn
    image: mysql:latest
    command: ['mysql']
    args: ['-c', 'show databases;']
    env:
    - name: MYSQL_HOST
      valueFrom:
        secretKeyRef:
          name: production-rds-conn-string
          key: endpoint
    - name: MYSQL_PWD
      valueFrom:
        secretKeyRef:
          name: production-rds-conn-string
          key: password
    - name: USER
      valueFrom:
        secretKeyRef:
          name: production-rds-conn-string
          key: username
```

```
    - name: MYSQL_TCP_PORT
      valueFrom:
        secretKeyRef:
          name: production-rds-conn-string
          key: port
```

aws-rds-connection-test.yaml hosted with ♡ by GitHub view raw

```
###Create a testpod that shows all the databases in the RDS Instance
kubectl apply -f aws-rds-connection-test.yaml
```

You should now see the databases in the logs of the Pod.

Throughout the article we have hardcoded the names, we also have an option to filter the resources using tags. But for some reason, my resources were not being filtered even after tagging them. If you were able to filter them using the tags please feel free to paste the solution in the comments section.

Until we meet again…….

Conclusion

Thanks for reading my article. Hope you have liked it. Here are some of my other articles that may interest you.

Recommended

Introduction to Jenkins Operator

Getting started with Jenkins Operator in Kubernetes

medium.com

Introduction to Bitnami Sealed Secrets

How to store your secrets in GitHub using Sealed Secrets and Kubese

medium.com

Introduction to External DNS in Kubernetes

How to automatically create DNS records in Kubernetes using External DNS

medium.com

Creating a GKE Cluster with GitHub Actions

Automating Kubernetes Cluster creation and Bootstrapping using GitHub Actions

medium.com

Reference

crossplane/crossplane

Crossplane, a Cloud Native Computing Foundation sandbox project, is an open-source Kubernetes add-on that extends any...

github.com

Crossplane Kubernetes AWS Cloud Computing Github

About Help Legal

Get the Medium app