Architecting a CI/CD pipeline for container and microservice-based applications

Alok
May 7, 2019 · 8 min read

I recently got the opportunity to engage with a customer to deliver a manageable, secure, scalable and highly available architecture using container orchestration

technology, for horizontal auto-scaling and a self-healing system.

We proposed architecture using AWS managed Kubernetes service (EKS) to easily deploy, manage, and scale containerized applications. The legacy DevOps pipeline consists of a CI/CD pipeline which includes a code repository (Bit Bucket), an automation server for continuous integration (Jenkins) for rolling out deployments.

While the Jenkins pipeline extends beyond CI to CD, the existing CI/CD automation was not leveraging the benefits of immutable infrastructure. Jenkins wasn't designed for deployment in Kubernetes and as the team was new to *kubectl* terminology there was a need for an easy to manage platform to make reliable deployments.

We rolled out a CI/CD pipeline using Jenkins and Spinnaker, an open source, multi-cloud continuous delivery platform to help the client deliver code with zero impact to customers. Like *Jon Snow of Winterfell*, Spinnaker proved to be the rightful heir streamlining the build and deploy process in Kubernetes.

This article will walk you through how spinnaker is integrated seamlessly with various tools and services automating the deployment for a sample application. You can explore more about Spinnaker and its features here https://www.spinnaker.io/ .

> ***"It's when you set your spinnaker that you're ready to fly with the wind."***

Installing Spinnaker

There are several ways to install Spinnaker. We will be deploying and configuring Spinnaker with a command-line administration tool called Halyard. (you can refer to the official installation guide: https://www.spinnaker.io/setup/install/). Here are a few requirements to move ahead with this deployment.

1. A K8s cluster with enough resources to support a deployment of around 10 pods.
2. A Jenkins server with access to AWS ECR registry to push docker images.

First, let's create a dedicated spinnaker namespace

```
kubectl create ns spinnaker
```

Second, let's create a halyard deployment to configure Spinnaker. Please feel free to use the latest stable release of Halyard.

```
kubectl create deployment hal --image gcr.io/spinnaker-
marketplace/halyard:1.15.0
```

We will create a service account with cluster admin role and add this account to halyard deployment spec.

```
kubectl create -f
https://raw.githubusercontent.com/SystemicEmotions/demoapp/master/sp
innaker/accnt.yaml
```

Edit the halyard deployment to add the service account as shown below:

```
spec:
  serviceAccountName: spinnaker-service-account
  containers:
  - image: gcr.io/spinnaker-marketplace/halyard:1.15.0
```

Once the halyard pod is up and running, let’s start a shell session to run halyard commands

```
kubectl exec -it <halyard-pod> bash
```

Configuring persistence storage

We will configure S3 for persisting Application settings and configured Pipelines in Spinnaker. We will need to create a bucket and a directory in it to specify as below.

```
hal config storage s3 edit --access-key-id <access-id> \
--region <region> \
--bucket <bucket-name> \
--root-folder <spin-folder> \
--secret-access-key

hal config storage edit --type s3
```

Configuring container registry (ECR)

```
hal config provider docker-registry enable

hal config provider docker-registry account add my-ecr-registry \
--address ≤aws-ecr-address> \
--username AWS \
--password-command “aws --region us-east-1 ecr get-authorization-
token --output text --query ‘authorizationData[].authorizationToken’
| base64 -d | sed ‘s/^AWS://’”
```

Enabling Kubernetes provider and adding ECR registry

A Spinnaker account maps to a credential that can authenticate against the K8s Cluster. It also includes one or more docker registry accounts that are used as a source of images. Here, we will configure ECR registry mentioned in the previous step with the specified account.

```
hal config provider kubernetes enable
hal config provider kubernetes account add my-k8s-account --docker-
registries my-ecr-registry
hal config deploy edit --type distributed --account-name my-k8s-
account
```

Note the use of type distributed specified in the last command. In a distributed installation, Halyard deploys each of Spinnaker’s microservices separately. This is highly recommended for use in production.

Configuring Spinnaker to interact with the default k8s cluster

```
kubectl config set-cluster default --
server=https://kubernetes.default --certificate-
authority=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt

kubectl config set-context default --cluster=default

token=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
kubectl config set-credentials user --token=$token
kubectl config set-context default --user=user
kubectl config use-context default
```

Configuring Jenkins master to trigger build from Spinnaker

```
hal config ci jenkins enable
hal config ci jenkins master add my-jenkins-master \
--address <Jenkins URL> \
--username <admin> \
--password
```

Create services for Gate (API Gateway) and Deck (UI)

```
kubectl create -f
https://raw.githubusercontent.com/SystemicEmotions/demoapp/master/spinnaker/svcs.yaml
```

Keep a note of the load-balancer DNS endpoints

```
kubectl get svc -n spinnaker -o wide
```

We want to set the UI (Deck) to the EXTERNAL-DNS for port 9000, and the API (Gate) for the EXTERNAL-DNS for port 8084.

```
hal config security ui edit --override-base-url http://< deck –
external-IP>:9000
hal config security api edit --override-base-url http://< gate-
external-IP>:8084
```

Configuring Authentication Using an Identity Provider

Spinnaker supports multiple options for both authentication and authorization. We will use a SAML based IDP, Okta to implement single sign-on (SSO). Let's create an app in Okta and copy the metadata.xml. In order to create an app in Okta, we will define the following:

```
Single Sign On URL – http://<gate-endpoint>:8084/saml/SSO
Audience URI – spinnaker-test
NameID format – EmailAdress
```

Creating a Spinnaker app in Okta

Generate a keystore and key in a new Java Keystore with some password.

```
keytool -genkey -v -keystore saml.jks -alias saml -keyalg RSA -
keysize 2048 -validity 10000
```

Finally use below halyard command to configure authentication

```
hal config security authn saml edit \
--keystore /home/spinnaker/saml.jks \
--keystore-alias saml \
--keystore-password <mypass> \
--metadata /home/spinnaker/metadata.xml \
--issuer-id spinnaker-test \
--service-address-url http://<gate-endpoint>:8084
```

Select a spinnaker version and apply the deployment

```
hal version list
hal config version edit --version 1.x.x
```

Before running the final command, we can review the custom configurations using below command

```
hal config list
```

Finally deploy the selected version of spinnaker.

```
hal deploy apply
```

Configuring the Build in Jenkins

Before we are able to deploy an application from Spinnaker, we would need to create a pipeline to build the image and push it to ECR from Jenkins.

We will be using the below repository for a sample deployment.

https://github.com/SystemicEmotions/demoapp

Below are the contents of the repository

Dockerfile — to dockerize a static html file using nginx as base image.

Jenkinsfile — this defines the stages in Jenkins: Clone repo->Build image->Test image->Push image to ECR

Index.html — a static html page

I have defined a webhook for triggering the spinnaker pipeline whenever any code changes are committed to the github repository.

Defining a webhook in github repository

Create a pipeline project in Jenkins and define the repository URL and script Path.

Creating a pipeline project in Jenkins

Also, we will need to create credentials for ECR in Jenkins and modify the “Push image” stage in Jenkinsfile to use that credentials.

```
stage(‘Push image’) {
docker.withRegistry(‘https://xxxxxxxxx.dkr.ecr.us-east-
1.amazonaws.com', ‘ecr:us-east-1:aws_ecr_cred’) {
app.push(“latest”)
}
}
```

Now, let’s go ahead and trigger a build manually.

Jenkins Job to build and push image to ECR

Once the pipeline is executed successfully, we should be able to see the docker image listed in ECR. We can now go ahead and start creating the pipeline for automating the entire build and deploy process in Spinnaker.

Create a server group in Spinnaker for application deployment

We will create a server group for application deployment. In order to access the webpage from the public network, we will first create a load balancer from Spinnaker console. Under Advanced Settings in Create new Load Balancer section, define type as Load Balancer.

Creating a load balancer from Spinnaker console

Once the load balancer is created, we will create a server group (set of pods) under the infrastructure section using the image that was built and pushed to ECR during the Jenkins build. Also, specify the load balancer that we have created in the previous step.

Creating a server group to deploy the image

Once the server group is created, we should be able to see running pods in our kubernetes cluster. We should now be able to access the sample webpage using the load balancer endpoint which should look something like below.

Sample web page deployed in kubernetes cluster

Creating a Pipeline for continuous deployment in Spinnaker

Now, let's build a pipeline in Spinnaker for continuous deployment. We will define 4 stages in this pipeline.

1. Configuration — this will include a trigger to get notified of any code changes in Github code repository.

Configuring a webhook to trigger pipeline

2. Jenkins — in this stage we will specify the Jenkins master and configure Jenkins job to trigger a build following stage 1.

Configuring Jenkins master to trigger Jenkins Job

3. Deploy — here we will use the existing template for application deployment configuration that we have specified while creating the server group at the beginning.

Creating a deployment configuration

4. Destroy Server Group — in this stage we will destroy our previous server group once our new pods are up and running.

Creating a final stage to destroy previous server group

Once we are done creating the pipeline, let's go try and commit a change to our index.html file in Github. Spinnaker pipeline will be triggered as soon as we commit

the changes in Github.

Spinnaker pipeline in progress

Once, all four stages are completely successful, we should be able to see the new changes getting reflected in our webpage. Here is a link to the video that I have recorded as a demo https://youtu.be/N4e3Zf1yFWY .

Your feedback is highly appreciated.

Docker Kubernetes Spinnaker Ci Cd Pipeline Continuous Deployment