GitOps ArgoCD project:

1. We create server and install Jenkins, docker, aws cli, terraform. Using terraform we build EKS cluster.
2. We create ECR repository
3. We setup CI job in Jenkins
4. We setup CD job in ArgoCD
5. We setup monitoring using Prometheus and Grafana
6. We setuo notifications using Prometheus alertmanager and Slack channel.

Step 1:

Create EC2 instance and install Jenkins, docker, aws cli, helm, terraform, kubectl

First we need to install Jenkins.

<https://www.jenkins.io/doc/tutorials/tutorial-for-installing-jenkins-on-AWS/>

Follow steps in below link to install docker in Amazon Linux.

<https://www.cyberciti.biz/faq/how-to-install-docker-on-amazon-linux-2/>

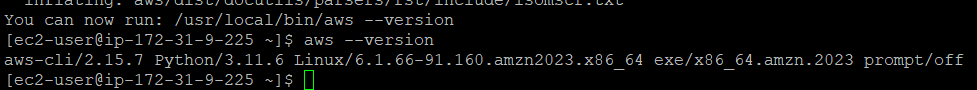
**Install AWS cli:**

curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"

sudo apt install unzip

unzip awscliv2.zip

sudo ./aws/install



Okay now after installing the AWS CLI, let's configure the AWS CLI so that it can authenticate and communicate with the AWS environment.

aws configure

A black screen with white text

Description automatically generated

## **Install and Setup Kubectl**

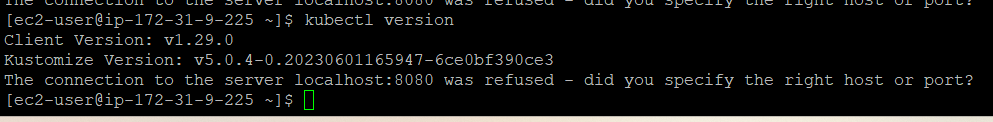
Moving forward now we need to set up the **[kubectl](https://kubernetes.io/docs/reference/kubectl/overview/" \t "_blank)** also onto the EC2 instance.

curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"

chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin

kubectl version

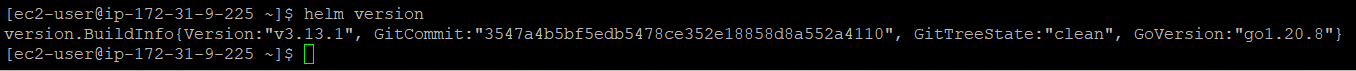


## **Install Helm chart**

$ curl -fsSL -o get\_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3

$ chmod 700 get\_helm.sh

$ ./get\_helm.sh



This way we install all AWS CLI, kubectl, eksctl and Helm.

**Install Terraform:**

Follow below steps to install terraform on AmazonLinux.

sudo yum install -y yum-utils shadow-utils

sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo

sudo yum -y install terraform

A black screen with white text

Description automatically generated

# **Creating an Amazon EKS cluster using terraform**

Code available in https://github.com/ksnithya/blue-green.git

git clone <https://github.com/ksnithya/blue-green.git>

cd blue-green

terraform init

terraform plan

terraform apply

aws eks --region ap-south-1 update-kubeconfig --name eks\_cluster\_demo

**Step 2:**

First we create repository in ECR.

Login to AWS console. Search for ECR.

ECR -> Get started.

Give name to repo. Then click on create.

A screenshot of a computer

Description automatically generated

A screenshot of a computer error message

Description automatically generated

Now repo will be created.

**A screenshot of a computer

Description automatically generated**

**Step 3:**

If we want to connect to ECR to our EC2 instance we need to attach “[AmazonEC2ContainerRegistryFullAccess](https://us-east-1.console.aws.amazon.com/iam/home?region=ap-south-1#/policies/details/arn%3Aaws%3Aiam%3A%3Aaws%3Apolicy%2FAmazonEC2ContainerRegistryFullAccess)” this policy. So we create one role and add this policy to it and attach to EC2 instance.

EC2 instance ->Action -> security -> Modify IAM role -> Select the role we need to attach -> click on “Update IAM role”.

Jenkins and Dockerfile code repo: <https://github.com/ksnithya/python-flask.git>

Deployment code repo: <https://github.com/ksnithya/eks-python-demo.git>

We will setup Jenkins job. We will create docker image and push to AWS ECR.

To connect ECR using Jenkins we need to install “Amazon ECR” plugin in Jenkins.

Dashboard -> Manage Jenkins -> Plugins -> Available Plugins -> search “Amazon ECR”.

A screenshot of a chat

Description automatically generated

Also we need to install docker plugins. Docker, Docker Pipeline.

To create the job click on

Dashboard -> New Item -> Give job name -> select “pipeline”-> ok

We create a declarative pipeline. First we checkout out git repository. We can generate the checkout syntax from “Pipelinesyntax”. Click on “Pipeline Syntax”.

A white background with black text

Description automatically generated

In sample step select “Checkout: Checkout from version control”. Then give our github repository URL we are going to use.

A screenshot of a computer

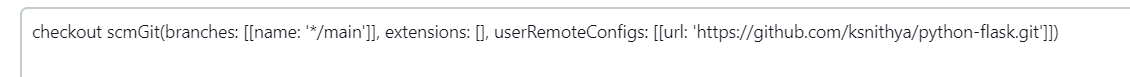
Description automatically generated

Type the branch name we want to use. Then click on generate.

A screenshot of a chat

Description automatically generated

We will get the required output. We can use the same in our pipeline code.



pipeline{

agent any

stages{

stage("Check out"){

steps{

checkout scmGit(branches: [[name: '\*/main']], extensions: [], userRemoteConfigs: [[url: 'https://github.com/ksnithya/python-flask.git']])

}

}

Then we create our “Build stage”. We will define our repo url as environment variable so that we can use is anywhere in the pipeline.

pipeline{

agent any

environment{

Region = "ap-south-1"

Name = "python-flask-demo"

TAG = "v1"

VERSION = "${env.BUILD\_ID}"

registry="822626997628.dkr.ecr.ap-south-1.amazonaws.com"

}

stages{

stage("Check out"){

steps{

checkout scmGit(branches: [[name: '\*/main']], extensions: [], userRemoteConfigs: [[url: 'https://github.com/ksnithya/python-flask.git']])

}

}

**stage("Build Docker Image"){**

**steps{**

**script{**

**echo "Building ${Name} image"**

**sh 'docker build -t ${Name}:${TAG} .'**

**sh 'docker tag ${Name}:${TAG} ${registry}/${Name}:${VERSION}'**

**}**

**}**

**}**

**Now we** push our image to ECR.

**stage("Push Im age to ECR"){**

**steps{**

**script{**

**sh 'aws ecr get-login-password --region ${Region} | docker login --username AWS --password-stdin ${registry}'**

**sh 'docker push ${registry}/${Name}:${VERSION}'**

**}**

**}**

**}**

Then, now we clone the repo where our python-demo manifest files exist to update the image name in deploy.yml file.

**stage('Clone/Pull Repo') {**

**steps {**

**script {**

**if (fileExists('eks-python-demo')) {**

**echo 'Cloned repo already exists - Pulling latest changes'**

**dir("eks-python-demo") {**

**sh 'git pull'**

**}**

**} else {**

**echo 'Repo does not exists - Cloning the repo'**

**sh 'git clone -b feature https://github.com/ksnithya/eks-python-demo.git'**

**dir("eks-python-demo"){**

**sh "ls -l"**

**}**

**}**

**}**

**}**

**}**

Now we update the manifest deploy.yml file with new image.

**stage('Update Manifest') {**

**steps {**

**dir("eks-python-demo") {**

**sh 'sed -i "s|image: .\*$|image: ${registry}/${Name}:${VERSION}|" deploy.yml'**

**sh 'cat deploy.yml'**

**}**

**}**

**}**

**}**

Now we push back the update to feature branch in our repo. Since we are pushing the data to repo it require username and password of github. To dom that we create token in github and we add that in credentials in Jenkins and we use that.

How to create token:

Login to github.

Settings -> Developer settings -> Personal access token -> Fine grained token.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

It will give one token. Save that in safe place. We cant get the token again. We need to regenerate it.

Now we add that token in Jenkins credentials. Login to Jenkins.

Dashboard -> Manage Jenkins -> credentials -> Global -> Add credentials.

Add the user name as github id and password field add “token” and save it. We add credentials in environments variable and use it.

pipeline{

agent any

environment{

Region = "ap-south-1"

Name = "python-flask-demo"

TAG = "v1"

VERSION = "${env.BUILD\_ID}"

registry="822626997628.dkr.ecr.ap-south-1.amazonaws.com"

**github\_token = credentials('github-token')**

}

stage('Commit & Push') {

steps {

dir("eks-python-demo") {

sh "git config --global user.email 'ksnithyamsc@gmail.com'"

sh 'git remote set-url origin https**://$github\_token**@github.com/ksnithya/eks-python-demo.git'

sh 'git checkout feature'

sh 'git add -A'

sh 'git commit -am "Updated image version for Build - $VERSION"'

sh 'git push origin feature'

}

}

}

Now we raise PR request to pull the code to main branch.

stage('Merge Request') {

steps {

dir("eks-python-demo") {

sh "git config --global user.email 'ksnithyamsc@gmail.com'"

sh 'git remote set-url origin https://$github\_token@github.com/ksnithya/eks-python-demo.git'

sh 'git checkout feature'

// Prepare main branch

sh 'git fetch --all'

sh 'git checkout main'

sh 'git pull origin main'

// Merge feature into main

sh 'git merge -m "merging to main branch" origin/feature'

// Push changes

sh 'git push origin main'

}

}

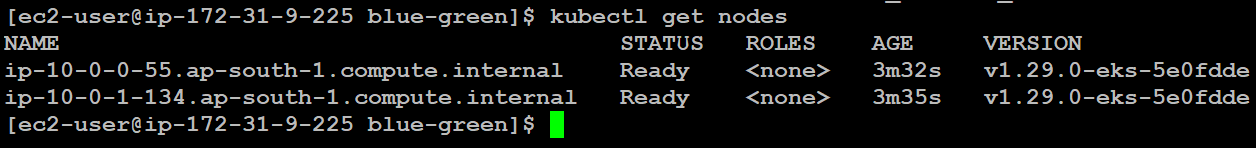
}

Finally we have completed out CI Job using Jenkins.

**Step 4:**

Now we create CD job using ArgoCD. For this we need running EKS cluster. We have already created EKS cluster in Step 1.

Terraform code: <https://github.com/ksnithya/EKS-Terraform.git>



First we create namespace to install argocd.

kubectl create namespace argocd

A screenshot of a computer

Description automatically generated

Then run the below command to create ArgoCD setup.

kubectl apply -n argocd -f <https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml>

It will create all resources required for ArgoCD.

Kubectl get -n argocd all

A screenshot of a computer program

Description automatically generated

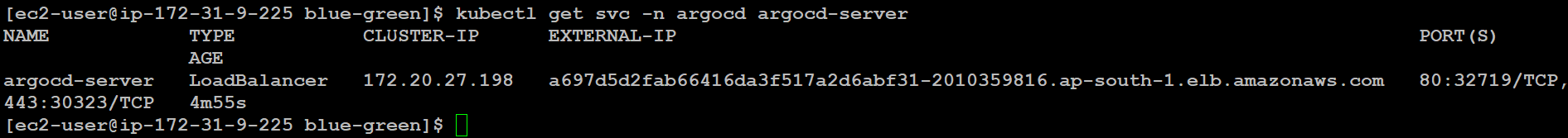
To access the argocd from UI we need to change the service of argocd-server to Nodeport/Loadbalancer. We will change to Loadbalancer.

A screen shot of a computer

Description automatically generated

kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'

We can access using the external ip dns name.



Default username is “admin” and password we can get it from secret.

Argocd-initial-admin-secret contains the password, We van convert that to base64.

echo “<password>”|base64 -d.

A screenshot of a computer program

Description automatically generated

We need to change the password after login.

A screenshot of a computer

Description automatically generated

Now we install argocd CLI.

**sudo curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/latest/download/argocd-linux-amd64  
sudo chmod +x /usr/local/bin/argocd**

A computer screen with white text

Description automatically generated

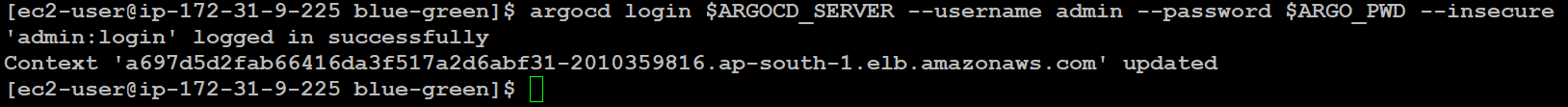
Now we login to argocd .

We can set server name, password as environmenet variable.

export ARGOCD\_SERVER=<ip address/dns name of argocd-server service>

export ARGO\_PWD=<password of argocd login>

argocd login $ARGOCD\_SERVER --username admin --password $ARGO\_PWD –insecure



Now we start creating our application.

For that we add our repo into it.

A screenshot of a computer

Description automatically generated

A screenshot of a phone

Description automatically generated

Select the connection method as “Https”, Then project as “default”. We can also use different project if we have created one. Then give our repo URL. Then click on connect.

A screenshot of a computer

Description automatically generated

A close-up of a screen

Description automatically generated

Now we start creating our application.

Applications -> New app

A screenshot of a computer

Description automatically generated

Fill the below details,

A screenshot of a computer

Description automatically generated

We can choose the branch name also. By default it is main/master. If you want to change you need to change the revision to branch name. For checking purpose I have changed the image tag to 5 in feature branch repo.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

We can give the namespace where we want to crate our application.

A white rectangular object with blue lines

Description automatically generated

Now app is created.

A screenshot of a computer

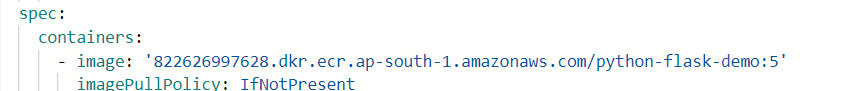
Description automatically generated

Then click on sync. App will be synced and deployed.

A screenshot of a computer

Description automatically generated

In deployment image name tag is 5.



Our application created in the namespace we have given.

A screen shot of a computer

Description automatically generated

Now we create one more app for main/master branch same as above steps.

In our main branch tag is 6.

A computer screen shot

Description automatically generated

A screenshot of a computer

Description automatically generated

A close-up of a computer code

Description automatically generated

Now our application is running.

A screenshot of a computer error message

Description automatically generated

Now I will modify the python application in our repository. Application will automatically deployed completely.

A screenshot of a computer

Description automatically generated

**Step 5:**

**Steps link:** [**https://argo-cd.readthedocs.io/en/stable/operator-manual/metrics/#prometheus-operator**](https://argo-cd.readthedocs.io/en/stable/operator-manual/metrics/#prometheus-operator)

Now, We setup monitoring using Prometheus and Grafana.

First we install Prometheus using helm.

Add Prometheus helm chart repository

helm repo add prometheus-community <https://prometheus-community.github.io/helm-charts>

Update the helm chart repository

helm repo update

helm repo list

A computer screen with white text

Description automatically generated

Create prometheus namespace

kubectl create namespace prometheus

Install Prometheus

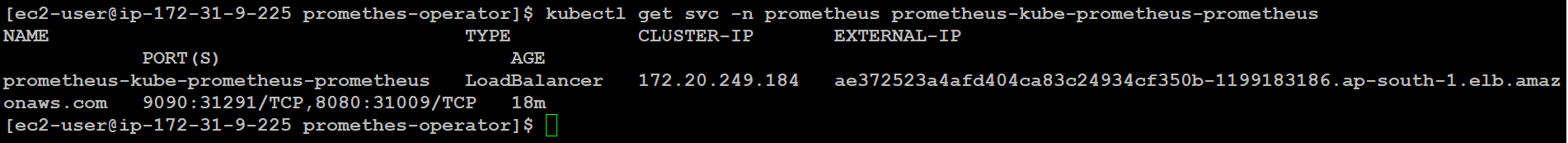
helm install prometheus prometheus-community/kube-prometheus-stack -n Prometheus.

A black screen with white text

Description automatically generated

To access the Prometheus outside we need to change the svc to Loadbalance/nodeport.

kubectl patch svc prometheus-kube-prometheus-prometheus -n prometheus -p '{"spec": {"type": "LoadBalancer"}}'



Prometheus listen on port 9090.

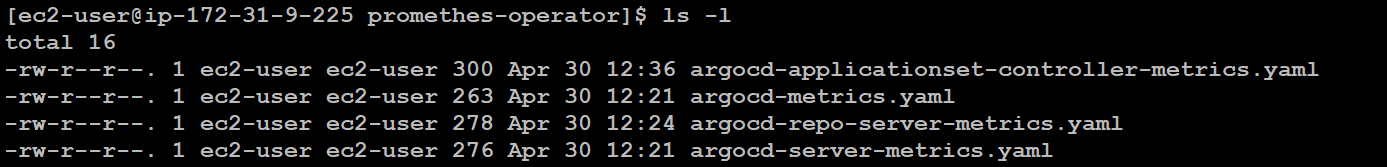
Now we can access using http:// ae372523a4afd404ca83c24934cf350b-1199183186.ap-south-1.elb.amazonaws.com:9090

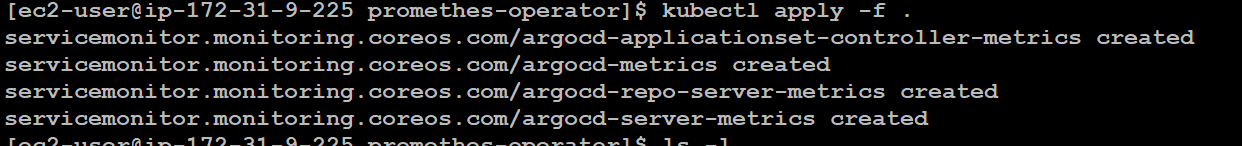
Then we setup the Prometheus operator.

If using Prometheus Operator, the following ServiceMonitor example manifests can be used. Add a namespace where Argo CD is installed and change **metadata.labels.release** to the name of label selected by your Prometheus.

Create the ArgoCD service monitors below within the argocd namespace.  
  
  
   1. argocd-metrics  
  
   2. argocd-server-metrics  
  
   3. argocd-repo-server-metrics  
  
   4. argocd-applicationset-controller-metrics

File repo: <https://github.com/ksnithya/prometheus-operator.git>





Now we can see argocd metrics in target.

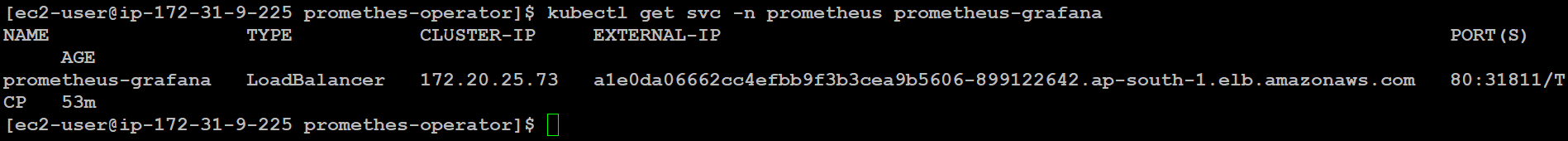
A screenshot of a computer

Description automatically generated

Now we setup Grafana to view in graphical mode.

By default it will be installed along with Prometheus. We need to change the service to Loadbalancer/nodeport so that we can access from outside.

kubectl patch svc prometheus-grafana -n prometheus -p '{"spec": {"type": "LoadBalancer"}}'



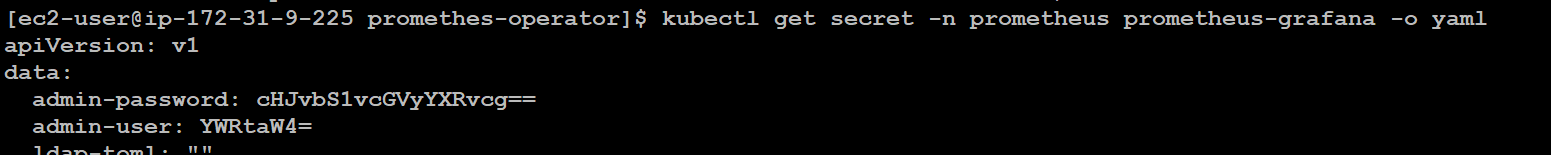
Now we can access using IP/DNS of svc. Grafana uses 80 port.

http://<grafana dns>:80

Default usename is admin. Password we can get it from secrets.

A computer screen with white text

Description automatically generated promethes-operator]$ kubectl get secret -n prometheus prometheus-grafana -o yaml

A screenshot of a login screen

Description automatically generated

ClickDashboard->new->Import

A black screen with a black background

Description automatically generated with medium confidence

We can use already existing dashboard of argocd.Id is “14584”.

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Now we enable alertmanager for argocd.

In Prometheus UI pageclick on status->Rules.

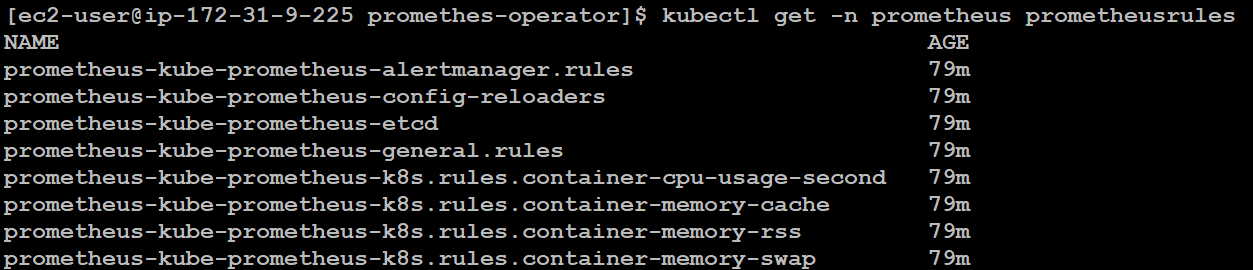
We can see all rules which are setup. Here we are going to add argocd rules.

A screenshot of a computer

Description automatically generated

For that we need to edit “prometheus-kube-prometheus-alertmanager.rules” rulesin Prometheus rules.

kubectl get -n prometheus prometheusrules



kubectl edit -n prometheus prometheusrules prometheus-kube-prometheus-alertmanager.rules

Append the below yaml snippet at under groups: and save (fix the indentation if needed):

- name: ArgoCD Rules

rules:

- alert: ArgoApplicationOutOfSync

expr: argocd\_app\_info{sync\_status="OutOfSync"} == 1

for: 5m

labels:

severity: warning

annotations:

summary: "'{{ $labels.name }}' Application has

synchronization issue"

A computer screen with white text

Description automatically generatedIn Prometheus UI page click on status->rules. We can see our new rule added.

A screenshot of a computer

Description automatically generated

All application are sync state. Our rule is in green.

A screenshot of a computer

Description automatically generatedNow we redploy the application.It is in “outofsync”. Now alert will be generated.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

We sync the appmanually after that alert will change to “green”.

A screenshot of a computer

Description automatically generated

**Step 6:**

[**https://argo-cd.readthedocs.io/en/stable/operator-manual/notifications/services/slack/**](https://argo-cd.readthedocs.io/en/stable/operator-manual/notifications/services/slack/)

Now we setup slack notification.

First we need to create a slack app.

Login to <https://api.slack.com/apps.Then> click on “create app”.

A screenshot of a computer

Description automatically generated

Give name and select the workspace.

A screenshot of a computer

Description automatically generated

Then click on “Oauth & Permissions”.

A screenshot of a computer

Description automatically generated

Under scope add below permissions,

A screenshot of a computer

Description automatically generated

After settingup the scope install it on workspace.

A screenshot of a computer

Description automatically generated

Now allow the permissions shown.

A screenshot of a computer

Description automatically generated

Copy the Bot User OAuth Token and keep it saved for later use in the Argo CD Notifications configurations.

A black pen on a white background

Description automatically generated

Then login to slack channel.

<https://app.slack.com/client/T06KUE7UKTP/C071QSC3FH7?geocode=en-in>

Then create one channel. Channel -> create ->create channel

A screenshot of a computer

Description automatically generated

Give channel name and create it.

A screenshot of a video chat

Description automatically generated

A screenshot of a computer

Description automatically generated

Now we add our app to channel. Type @. It will display all app. Select Argocd-Notification and press enter.

A screenshot of a chat

Description automatically generated

A blue rectangle with black text

Description automatically generated

We already installed argocd notification and secret in cluster.

kubectl get -n argocd all|grep -i notification

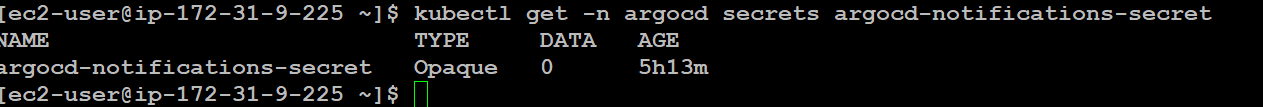
kubectl get -n argocd secrets

A screen shot of a computer

Description automatically generated

It is empty secret. We add data into it.

kubectl get -n argocd secrets argocd-notifications-secret

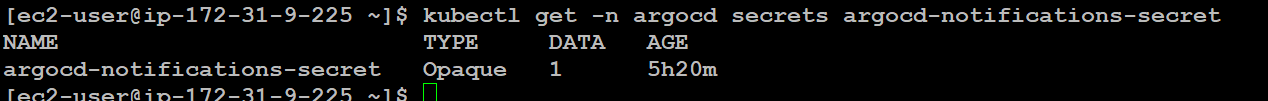


kubectl edit -n argocd secrets argocd-notifications-secret

A number on a black background

Description automatically generated

Now one data is added.



Now we edit the configmap in argocd and add the slack channel details.

A screen shot of a computer

Description automatically generated

kubectl edit cm -n argocd argocd-notifications-cm A computer screen with white text

Description automatically generated

data:

service.slack: |

token: $slack-token

template.app-sync-succeeded-slack: |

message: |

Application {{.app.metadata.name}} is now {{.app.status.sync.status}}

trigger.on-sync-succeeded: |

- when: app.status.sync.status == 'Synced'

send: [app-sync-succeeded-slack]

Now we are going edit the application and add the annotation to it.

Add annotation in application yaml file to enable notifications for specific argocd app. The following example uses the [on-sync-succeeded trigger](https://argo-cd.readthedocs.io/en/stable/operator-manual/notifications/catalog/#triggers):

kubectl get appproj -n argocd

A black background with white text

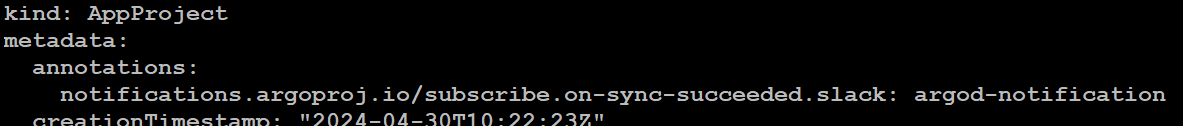
Description automatically generated

annotations:

notifications.argoproj.io/subscribe.on-sync-succeeded.slack: my\_channel

notifications.argoproj.io/subscribe.<trigger name> : channelname

kubectl edit appproj -n argocd default



metadata:

annotations:

notifications.argoproj.io/subscribe.on-sync-succeeded.slack: argod-notification

Note:

Above notification will be send when application of argocd is synced. Like this we can add many notifications.

<https://argo-cd.readthedocs.io/en/stable/operator-manual/notifications/services/slack/>

A screenshot of a chat

Description automatically generated