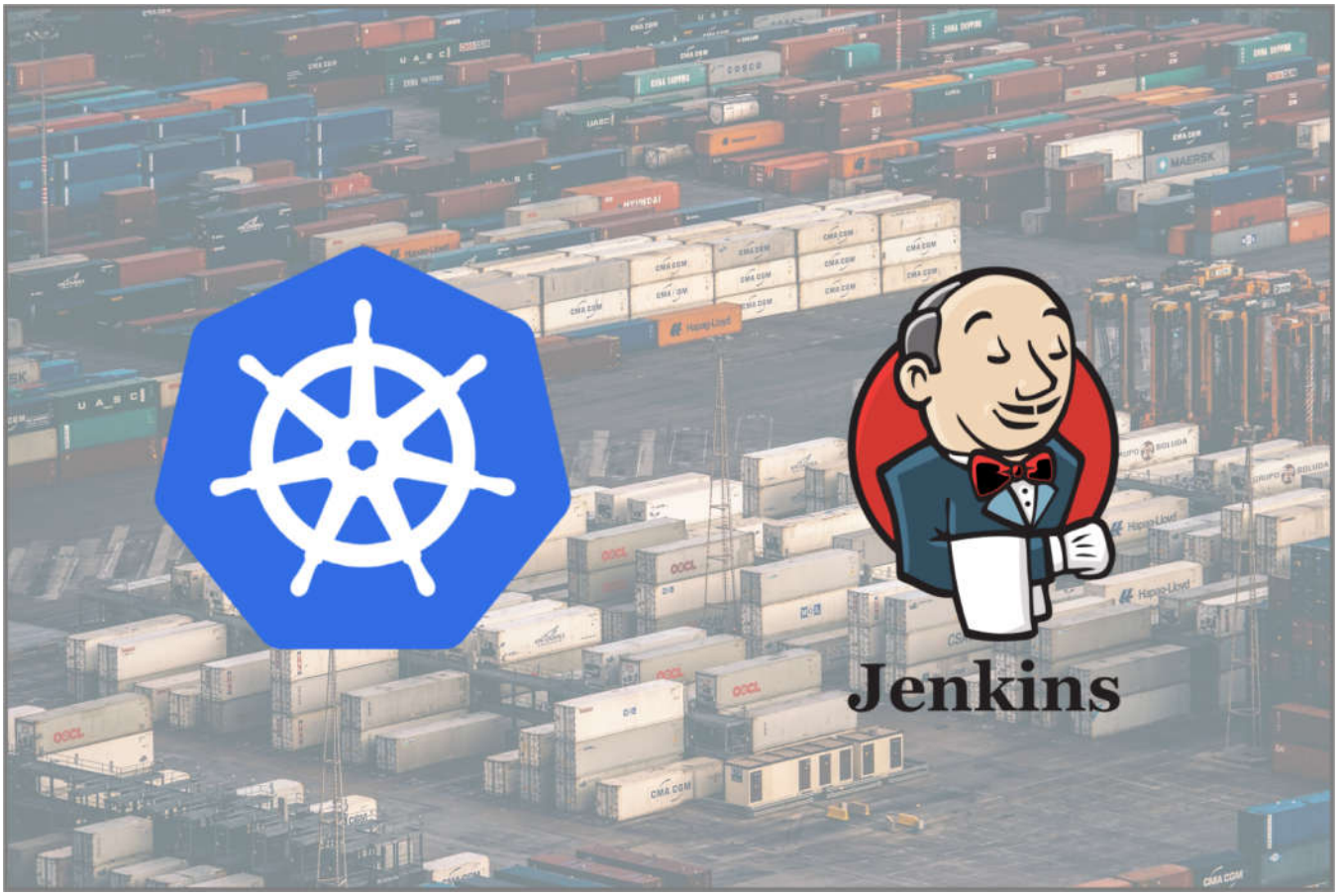
Configuring CI/CD on Kubernetes with Jenkins

Containerum
Sep 7, 2018 · 9 min read



by Alexander Kirillov

The software industry is rapidly seeing the value of using containers as a way to facilitate development, deployment, and environment orchestration for application developers. That's because containers effectively manage environmental differences, allow for improved scalability, and provide predictability that supports Continuous Delivery (CD) of new features. In addition to the technical advantages, containers have been shown to dramatically reduce the cost model of complex environments.

Large-scale and highly-elastic applications that are built in containers definitely have their benefits, but managing the environment can be daunting. This is where

an orchestration tool like Kubernetes really shines.

Kubernetes is a platform-agnostic container orchestration tool created by Google and heavily supported by the open source community as a project of the Cloud Native Computing Foundation. It allows you to spin up the number of container instances and manage them for scaling and fault tolerance. It also handles a wide range of management activities that would otherwise require separate solutions or custom code, including request routing, container discovery, health checks, and rolling updates.

Kubernetes is compatible with the majority of CI/CD tools which allows developers run tests, deploy builds in Kubernetes and update applications with no downtime. One of the most popular CI/CD tools now is Jenkins, and this article will focus on configuring a CI/CD pipeline with Jenkins and Helm on Kubernetes.

Why Jenkins is becoming the CI/CD tool of choice for more and more DevOps

There are several reasons why Jenkins is gaining momentum. First, it is open source and free. Second, it is user-friendly, easy to install and does not require additional installations or components.

Jenkins is also quite easy to configure, modify and extend. It deploys code instantly, generates test reports. Jenkins can be configured according to the requirements for continuous integrations and continuous delivery.

Jenkins is available for all platforms and different operating systems, whether it is OS X, Windows or Linux. It also boasts rich plugin ecosystem. The extensive pool of plugins makes Jenkins flexible and allows building, deploying and automating across various platforms.

Since it is open source, there is no shortage of support from large online communities of agile teams. Finally, most of the integration work is automated. Hence fewer integration issues. This saves both time and money over the lifespan of a project.

CI/CD Steps

CI/CD process with Jenkins generally follows the following scheme:

- Checkout Code
- Run Unit Tests
- Dockerize App
- Push dockerized app to Docker Registry
- Use Ansible Playbook to deploy the dockerized app on K8s

To see how it works, let's start with Jenkins installation. We will use a machine with CentOS 7 with Docker and Kubernetes installed.

Installing Jenkins

Step 1: Update your CentOS 7 system

```
sudo yum install epel-release nodejs
sudo yum update
```

Step 2: Install Java

```
sudo yum install java-1.8.0-openjdk.x86_64
sudo cp /etc/profile /etc/profile_backup
echo 'export JAVA_HOME=/usr/lib/jvm/jre-1.8.0-openjdk' | sudo tee
-a /etc/profile
echo 'export JRE_HOME=/usr/lib/jvm/jre' | sudo tee -a /etc/profile
source /etc/profile
```

Step 3: Install Jenkins

```
sudo rpm --import https://pkg.jenkins.io/redhat-
stable/jenkins.io.key
wget -O /etc/yum.repos.d/jenkins.repo
https://pkg.jenkins.io/redhat-stable/jenkins.repo
sudo yum install -y jenkins
```

Step 4: Start Jenkins and check if it is running:

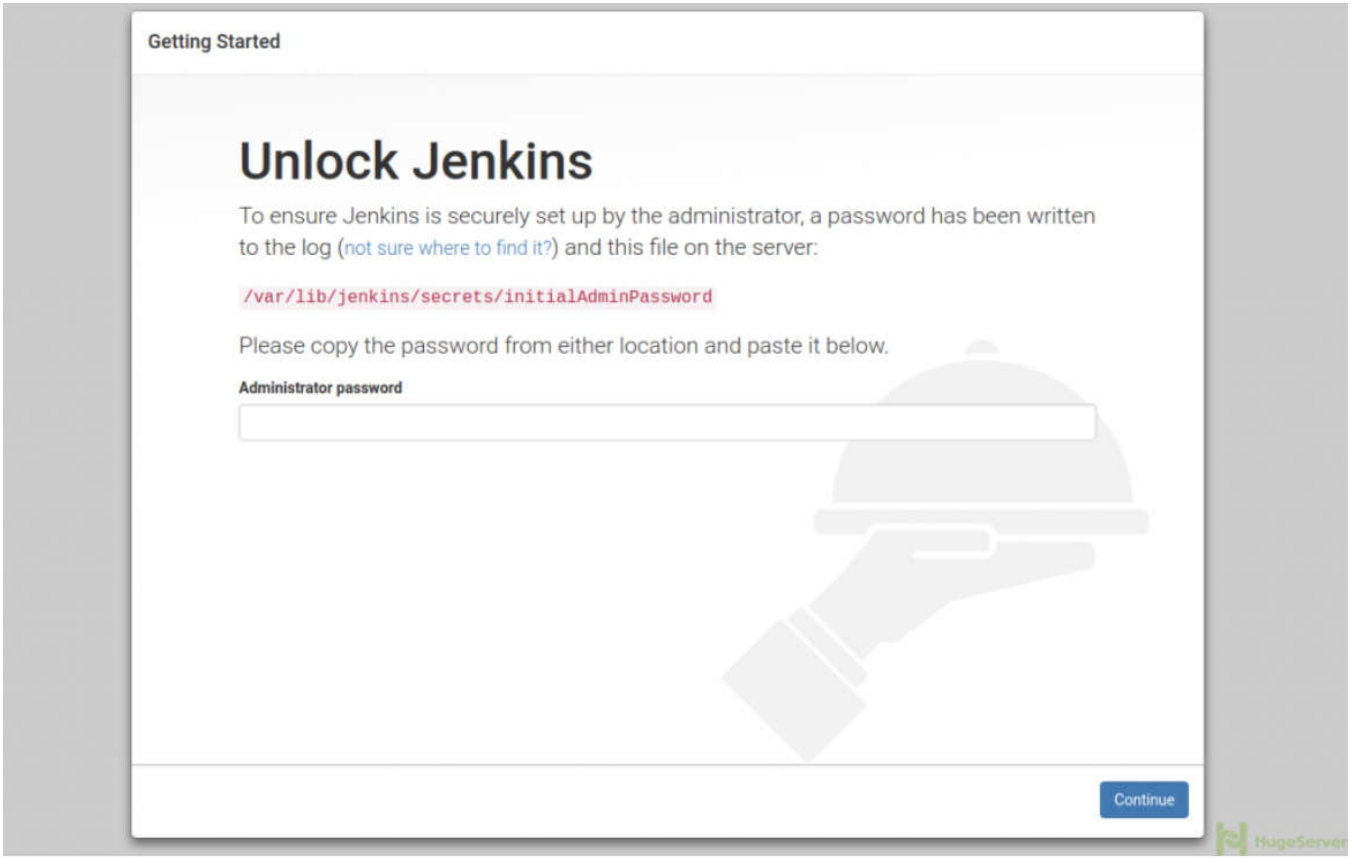
```
sudo systemctl start jenkins
sudo systemctl status jenkins
```

Step 5: Set up Jenkins

To start setting up Jenkins, we need to visit its web dashboard running on port 8080. Open your browser and see your public IP address or your Domain name followed by the port number through it:

```
http://YOUR_IP_OR_DOMAIN:8080
```

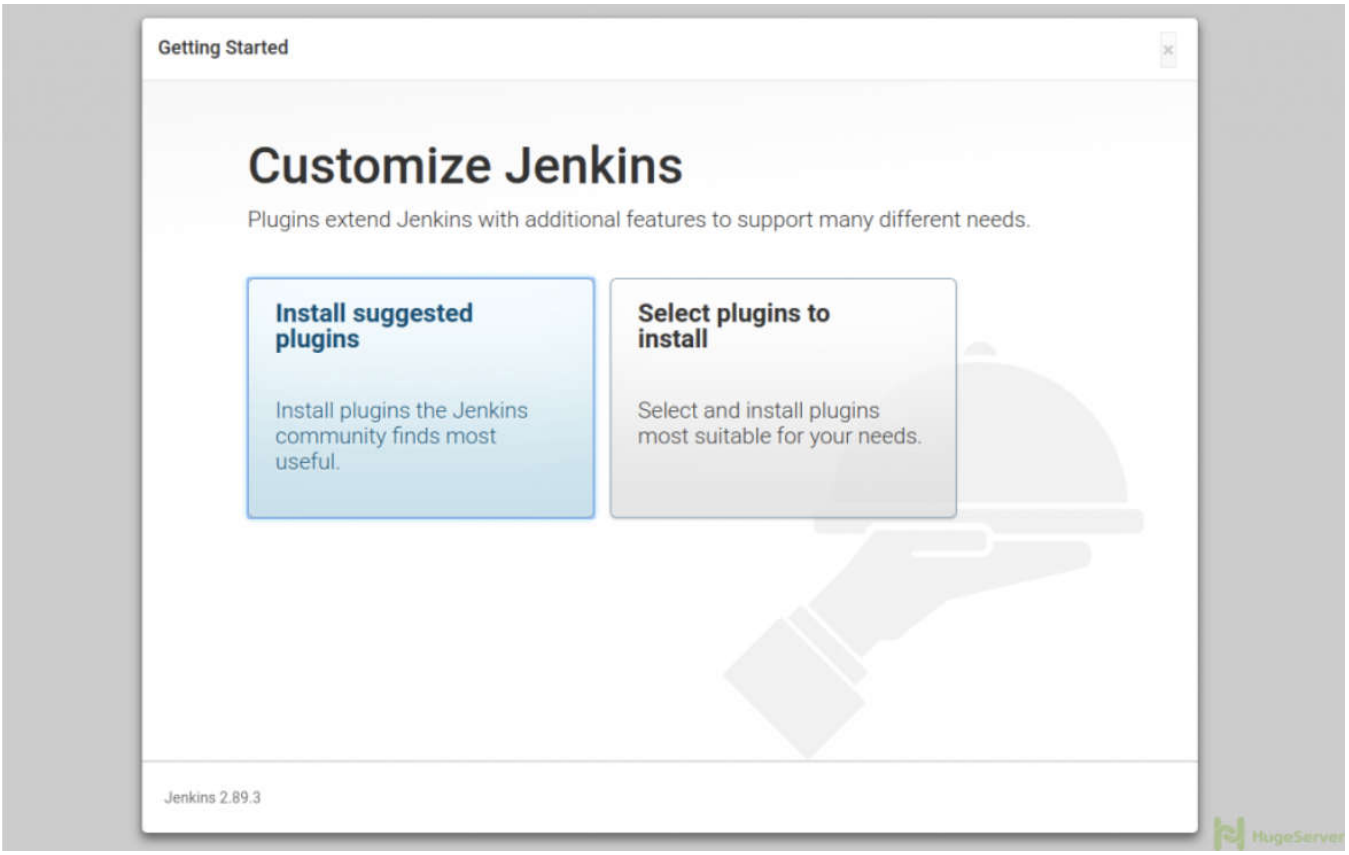
You will see a page like the one below:



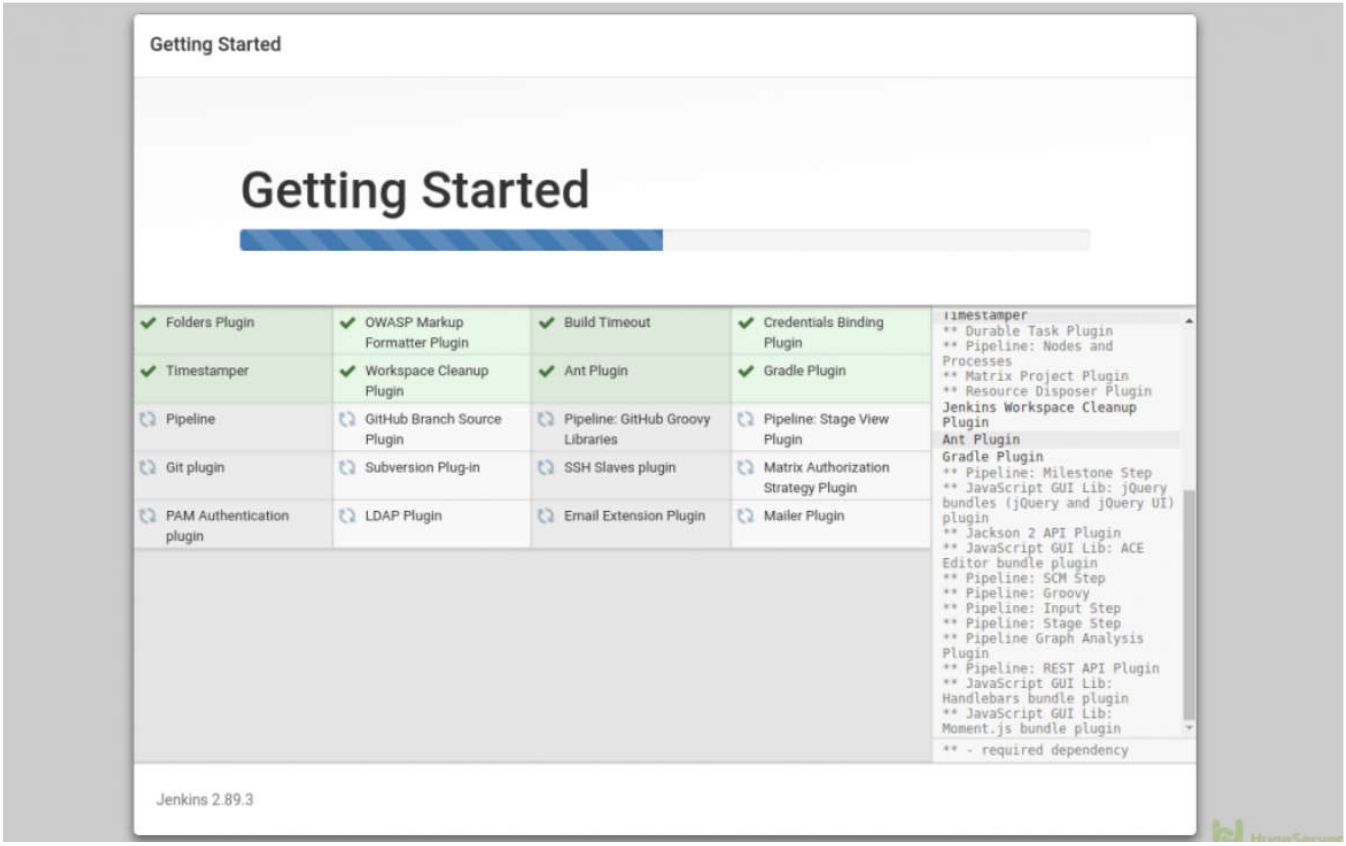
To get a password, run:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

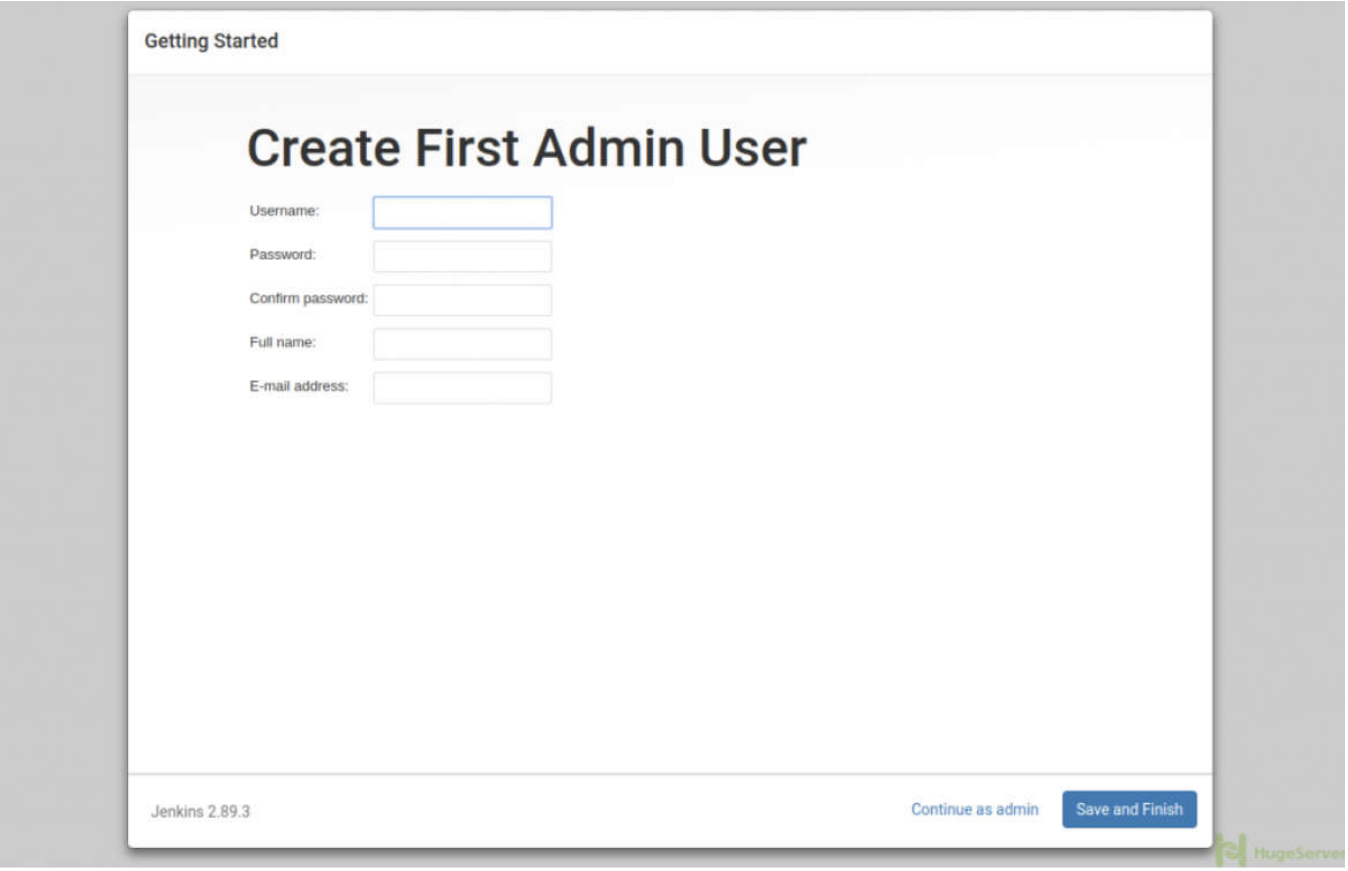
Paste the password in the “Administrator password” field and hit continue to see the following page:



If you are new to Jenkins, we recommend you to select the ‘Install suggested plugins’. Now you can see that Jenkins is installing some plugins:



After that, you will de directed to a page where you have to create your first admin user:



Preparing Jenkins Server

Jenkins offers a simple way to set up continuous integration and continuous delivery environment for almost any combination of languages and source code repositories. Let's configure Jenkins Server, which consists of installing Docker, Ansible, Helm, and Docker plugins.

Configuring Docker

Docker is hotter than hot because it makes it possible to get far more apps running on the same old servers and it also makes it very easy to package and ship programs. Create jenkins user to docker group:

```
$ sudo groupadd docker
$ sudo usermod -aG docker jenkins
$ chmod root:docker /var/run/docker.sock
```

Go to `/etc/passwd`. Find j`enkins:x:996:993:Jenkins Automation Server:/var/lib/jenkins:/bin/false` and change it to `jenkins:x:996:993:Jenkins Automation Server:/var/lib/jenkins:/bin/bash`.

Added the ansible host to `/etc/ansible/hosts`:

```
[localhost]
127.0.0.1
```

Make sure the jenkins can access the cluster throgh kubectl:
`mv .kube/config to /var/lib/jenkins`

Finally, add jenkins user to sudo users with `$ visudo -f /etc/sudoers`. Add `jenkins ALL=NOPASSWD: ALL` to the file and save.

Installing Ansible

Ansible is an open source automation platform. It is very, very simple to setup and yet powerful. Ansible can help you with configuration management, application deployment, and task automation. It can also do IT orchestration, where you have to run tasks in sequence and create a chain of events to run on several different servers or devices.

```
$ sudo yum install python-pip
$ sudo pip install ansible
```

Installing and configuring Helm Package Manager

Helm helps you manage Kubernetes applications — with Helm Charts you can define, install, and upgrade even the most complex Kubernetes application.

```
$ wget https://storage.googleapis.com/kubernetes-
helm//var/lib/jenkins/ansible/sayarapp-deploy/deploy.yml-v2.8.1-
linux-amd64.tar.gz
$ tar -xzvf helm-v2.8.1-linux-amd64.tar.gz
$ sudo mv linux-amd64/helm /usr/local/bin
$ sudo -i -u jenkins
$ mkdir .kube ; $ touch .kube/config
```

→ copy the contents of /etc/kubernetes/admin.conf to~/.kube/config to access the cluster under user Jenkins if necessary. Then run:

```
$ helm init --upgrade
```

Installing Docker plugin on Jenkins

→ Docker plugin allows using docker host to dynamically provision build agents, run a single build, and then push an image to the registry.

Navigate to `http://your-ip:8080/pluginManager/available` and search for the plug-in "CloudBees Docker Build and Publish". Click Download Now and check the box to restart.

Creating the Pipeline on Jenkins

Go to Jenkins and select `New Item` on the left side, enter the name `POC` and select `pipeline` and click `ok`.

Generating Pipeline syntax for git and docker registry

The `Pipeline Syntax` section(`/job/PIPELINE/pipeline-syntax/`)will help you generate the Pipeline Script code which can be used to define various steps. Pick a step you are interested in from the list, configure it, click `Generate Pipeline Script`, and you will see a Pipeline Script statement that would call a step with that configuration.

Navigate to http://your-ip:8080/job/POC/pipeline-syntax/

Select git and provide the repo URL and user-name/password, if the repo is private, it will generate the syntax for you.

If you use DockerHub, select the withdocker-registry which we installed before and provide the credentials of the registry (`https://index.docker.io/v1/` for DockerHub). Click `Generate Pipeline Script` and you will get a script like that that you will use as credentials:

```
withDockerRegistry([credentialsId: '55d22be4-cff4-4609-a97d-
a74ad61ad12b', url: 'https://index.docker.io/v1/'])
```

Cloning a Helm chart

Clone a chart for our sample project:

```
$ sudo su - jenkins
$ mkdir ansible
$ git clone https://mAyman2612@bitbucket.org/mAyman2612/ci-cd-
k8s.git
$ cp -r ci-cd-k8s/ansible/sayarapp ansible/
```

→ This will clone a sample project with hello-world type of application. The Helm charts for our project are located at `ansible/sayarapp/templates` . You can replace the yamls with your own files for deployment and services. Here's the `deployment.yml` we will use:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-kubernetes
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello-kubernetes
  template:
    metadata:
      labels:
        app: hello-kubernetes
    spec:
```

```
      containers:
      - name: hello-kubernetes
        image: "{{ .Values.image.repository }}:{{
.Values.image.tag }}"
        ports:
        - containerPort: 3000
        imagePullPolicy: Always
```

And this is the `service.yml`:

```
apiVersion: v1
kind: Service
metadata:
  name: hello-kubernetes
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: hello-kubernetes
```

Configure Ansible to deploy Helm Chart

Let's create an Ansible playbook to call the helm chart

```
$ cp -r ci-cd-k8s/ansible/sayarapp-deploy /var/lib/jenkins/
```

The playbook we will use in this article looks like this :

```
- hosts: localhost
  vars:
    ImageName: ""
    Namespace: ""
    imageTag: ""
  #remote_user: ansible
  #become: true
  gather_facts: no
  connection: local
  tasks:
    - name: Create Namespace {{ Namespace }}
      command: "kubectl create namespace {{ Namespace }}"
      ignore_errors: yes
    - name: Deploy SayarApp
      command: "/usr/local/bin/helm install --name=sayar-{{
Namespace }}  --namespace={{ Namespace }} ../sayarapp --set
```

```
image.repository={{ ImageName }} --set image.tag={{ imageTag }} --
set namespace={{ Namespace }}"
      delegate_to: localhost
      ignore_errors: yes
    - name: Update SayarApp
      command: "/usr/local/bin/helm upgrade --wait --recreate-pods
--namespace={{ Namespace }} --set image.repository={{ ImageName }}
--set image.tag={{ imageTag }} --set namespace={{ Namespace }}
sayar-{{ Namespace }} ../sayarapp"
      delegate_to: localhost
      ignore_errors: yes
```

CI/CD with Jenkins Pipeline

The Jenkinsfile that we will use for the pipeline looks like this:

```
node{
  def Namespace = "default"
  def ImageName = "sayarapp/sayarapp"
  def Creds = "2dfd9d0d-a300-49ee-aaaf-0a3efcaa5279"
  try{
  stage('Checkout'){
      git 'https://mAyman2612@bitbucket.org/mAyman2612/ci-cd-
k8s.git'
      sh "git rev-parse --short HEAD > .git/commit-id"
      imageTag= readFile('.git/commit-id').trim()

}

stage('RUN Unit Tests'){
      sh "npm install"
      sh "npm test"
  }
  stage('Docker Build, Push'){
    withDockerRegistry([credentialsId: "${Creds}", url:
'https://index.docker.io/v1/']) {
      sh "docker build -t ${ImageName}:${imageTag} ."
      sh "docker push ${ImageName}"
        }

}
    stage('Deploy on K8s'){

sh "ansible-playbook /var/lib/jenkins/ansible/sayarapp-
deploy/deploy.yml  --user=jenkins --extra-vars
ImageName=${ImageName} --extra-vars imageTag=${imageTag} --extra-
vars Namespace=${Namespace}"
    }
     } catch (err) {
      currentBuild.result = 'FAILURE'
    }
}
```

Let's have a deeper look into the jenkinsfile.

Step 1 — Here we define some variables:

```
def Namespace = "default"
//default namespace on k84
def ImageName = "sayarapp/sayarapp"
// image name which will be pushed to docker registry
    def Creds = "2dfd9d0d-a300-49ee-aaaf-0a3efcaa5279"
// Creds of docker registry
```

Step 2 — Pull/Clone updates from our version control:

```
git 'https://mAyman2612@bitbucket.org/mAyman2612/ci-cd-k8s.git'
      sh "git rev-parse --short HEAD > .git/commit-id"
      imageTag= readFile('.git/commit-id').trim()
```

Step 3 — Run Unit Tests:

```
stage('RUN Unit Tests'){
         sh "npm install"
         sh "npm test"
    }
```

Step 4 — Docker Build and Push to docker registry

```
stage('Docker Build, Push'){
withDockerRegistry([credentialsId: "${Creds}", url:
'https://index.docker.io/v1/']) {
         sh "docker build -t ${ImageName}:${imageTag} ."
         sh "docker push ${ImageName}"
           }
      }
```

Step 5 — Call the Ansible playbook to deploy on K8s

```
stage('Deploy on K8s'){
sh "ansible-playbook ./ansible/sayarapp-deploy/deploy.yml  --
user=jenkins --extra-vars ImageName=${ImageName} --extra-vars
imageTag=${imageTag} --extra-vars Namespace=${Namespace}"
 }
```

Access the application running in Kubernetes:

```
$ kubectl get svc // to get the IP/Port of the application
```

Now curl http://<public-node-ip>:<node-port>.

Update the code

Now let's see if we got it right. Let's change our yaml files a little bit.

In `CI-CD-K8s/app/routes/root.js` change `hello K8s` to `update k8s` in line 3. Also, in `CI-CD-K8s/app/test/root.test.js` change `hello K8s` to `update k8s` in line 27.

```
module.exports = function(req, res, next) {
  res.contentType = "json";
  res.send({ message: "hello K8s" });
  next();
};
```

and

```
const chai = require("chai");
const sinon = require("sinon");
var rootResponder = require("../routes/root");

// const expect = chai.expect;
// const assert = chai.assert;
chai.should();
describe("Root Directory Test", function() {
  describe("Should Behave properly on GETing /", function() {
    const nextSpy = sinon.spy();
    const resSpy = { send: sinon.spy() };
    beforeEach(function() {
      nextSpy.resetHistory();
      resSpy.send.resetHistory();
    });
```

```
    it("should call next", function() {
      rootResponder({}, resSpy, nextSpy);
      nextSpy.calledOnce.should.be.true;
    });

    it("should call send on resp", function() {
      rootResponder({}, resSpy, nextSpy);
      resSpy.send.calledOnce.should.be.true;
    });
    it("should call send on resp with Hello World as a message",
function() {
      rootResponder({}, resSpy, nextSpy);
      resSpy.send.calledWith({ message: "hello K8s"
}).should.be.true;
    });
    it("should have json as the content type of the respones",
function() {
      rootResponder({}, resSpy, nextSpy);
      resSpy.contentType.should.exist;
      resSpy.contentType.should.equal("json");
    });
  });
});
```

Run the pipeline again and curl http://<public-node-ip>:<node-port>

Conclusion

We have demonstrated a simple CI/CD workflow with Jenkins, Docker, Ansible, Helm and Kubernetes. The main benefit of this stack is flexibility since it allows you to implement practically any type of workflow. The great thing is that this workflow can be extended or complexified depending on your development needs.

Thanks for reading! Don't forget to follow us on Twitter and join our Telegram chat to stay tuned!

Containerum Platform is an open source project for managing applications in Kubernetes available on GitHub. We are currently looking for community feedback, and invite everyone to test the platform! You can submit an issue, or just support the project by giving it a ☆. Let's make cloud management easier together!