1. adımdayız.

git checkout dev

git branch feature/msp-17

git checkout feature/msp-17

kubernetes için manifest dosyalarını yazmamız gerekiyor. biz bu kısımda Kompose isimli bir tool kullanacağız. bu tool elimizde bir docker-compose dosyası varsa bunu k8s yaml dosyalarına dönüştürüyor.

bazı parametreleri çevirmiyor; <https://kompose.io/>

build: dockerfile 
build; args 
build: cache_from 
cap_add 
cap_drop 
command 
config s 
con figs; short- 
syntax 
configs: long- 
syn tax 
cgroup_parent 
container_name 
credential spec 
deploy 
deploy. mode 
deploy. replicas 
deploy. 
placement 
v' 
Containers ecu r 
Container. SecurityContextCapabiIitiesDrop 
Contai rkr.Args 
Metadata.Name + 
Deployment.Spec.Containers.Name 
Deployrnent.SpecRepIicas / 
DeplovnentConfig.SpecRepIicas 
Affinity 
Only Create configMap 
If target path is', ignore this and or 
Not supported within Kubernetes_ S 
https://github.com/kubernetes/kub 
Only applicable to Windows contair 

bizim için toplamda 25-26 tane yaml dosyasını bizim için hazırlayacak bir tool Kompose.

docker-compose.yml dosyamızı k8s klasörü oluşturup onun altında oluşturalım.

dosya içeriği:

version: '3'

services:

config-server:

image: "{{ .Values.IMAGE\_TAG\_CONFIG\_SERVER }}"

ports:

- 8888:8888

labels:

kompose.image-pull-secret: "regcred"

image i bu şekilde neden yazıyrduk? helm kullanacağız, image isimlerini oradan çekecek.

kopmpose.image-pull-secret neden var? kompose docs tan alıyoruz:

kompose.image-pull-secret 
kuEærnetes secret name for imagePuIISecrets 

kubernetes secret name için gerekli. bütün server larda lazım olmuş. bütün podlar ECR dan image çekecek, credential gerekli. aws ecr login olup push pull ediyorduk. regcred isminde bir secret oluşturup bütün service ler gidip ECR dan image çekebilecek.

discovery-server:

image: "{{ .Values.IMAGE\_TAG\_DISCOVERY\_SERVER }}"

ports:

- 8761:8761

labels:

kompose.image-pull-secret: "regcred"

customers-service:

image: "{{ .Values.IMAGE\_TAG\_CUSTOMERS\_SERVICE }}"

deploy:

replicas: 2

ports:

- 8081:8081

labels:

kompose.image-pull-secret: "regcred"

visits-service:

image: "{{ .Values.IMAGE\_TAG\_VISITS\_SERVICE }}"

deploy:

replicas: 2

ports:

- 8082:8082

labels:

kompose.image-pull-secret: "regcred"

vets-service:

image: "{{ .Values.IMAGE\_TAG\_VETS\_SERVICE }}"

deploy:

replicas: 2

ports:

- 8083:8083

labels:

kompose.image-pull-secret: "regcred"

api-gateway:

image: "{{ .Values.IMAGE\_TAG\_API\_GATEWAY }}"

deploy:

replicas: 1

ports:

- 8080:8080

labels:

kompose.image-pull-secret: "regcred"

kompose.service.expose: "{{ .Values.DNS\_NAME }}"

kompose.service.type: "nodeport"

kompose.service.nodeport.port: "30001"

api-gateway de neden nodeport service tipini açıyoruz? neden dışarıdan ulaşmak istiyoruz? selenium testleri (functional) nin yapılması için uygulamanın dışarıya açıldığı service api-gateway olacak bu nedenle nodeprot ekliyoruz ki testler oradan yapılabilsin.

tracing-server:

image: openzipkin/zipkin

ports:

- 9411:9411

admin-server:

image: "{{ .Values.IMAGE\_TAG\_ADMIN\_SERVER }}"

ports:

- 9090:9090

labels:

kompose.image-pull-secret: "regcred"

hystrix-dashboard:

image: "{{ .Values.IMAGE\_TAG\_HYSTRIX\_DASHBOARD }}"

ports:

- 7979:7979

labels:

kompose.image-pull-secret: "regcred"

grafana-server:

image: "{{ .Values.IMAGE\_TAG\_GRAFANA\_SERVICE }}"

ports:

- 3000:3000

labels:

kompose.image-pull-secret: "regcred"

prometheus-server:

image: "{{ .Values.IMAGE\_TAG\_PROMETHEUS\_SERVICE }}"

ports:

- 9091:9090

labels:

kompose.image-pull-secret: "regcred"

mysql-server:

image: mysql:5.7.8

environment:

MYSQL\_ROOT\_PASSWORD: petclinic

MYSQL\_DATABASE: petclinic

ports:

- 3306:3306

mysql image inda regrecd yok çünkü docker hub dan gelecek image.

dosyayı kaydettik, kompose toolunu kuralım.

curl -L <https://github.com/kubernetes/kompose/releases/download/v1.28.0/kompose-linux-amd64> -o kompose

chmod +x kompose

sudo mv ./kompose /usr/local/bin/kompose

kompose version

helm kuralım:

curl <https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3> | bash

helm version

helm chart nasıl olutşruyorduk? chart oluşturduğumuzda bize default bir dosya yapısı oluştruuyordu. biz de manifesto dosyalarımız kuruyorduk.

K8s dizini içinde helm chart oluşturalım.

cd k8s

helm create petclinic\_chart

bizim lokalde oluşturduğumuz yapıya chart, paketleyip install ederek kullandığımıza ise release diyoruz.

v k8s 
petclinic chart 
> charts 
> templates 
.helmignore 
! Chart.yaml 
values.yaml 

chart ve içindeki dosyalar geldi.

charts klasörü içinde bu chartın ihtiyaç duyduğuğ başka chart ve dependencies varsa burada tutulur.

templates manifest dosyalarını tutar.

chart.yaml chartla ilgili version ve diğer bilgileri tutuyor.

values.yaml da manifest dosyalarında değişebilecek veriler için değişken oluşturup buradan değiştirmek için kullanılır.

rm -r petclinic\_chart/templates/\*

ile templates altındaki tüm dosyaları siliyoruz.

Artık kompose tool unu kullanabiliriz. docker-compose.yml dosyasına kuberntese çevrilebilecek parametreleri ekliyoruz. gerekli label ları ekliyoruz.

labels: 
kompose . image-pull-secret: "regcred " 

özel bir repodan image çekeceksek credentialını secret olarak düzenleyip ismini bu etiketle tanıtıyoruz.

api -gateway : 
Image : 
deploy : 
replicas : 
ports : 
. Values.1MAGE TAG API_GATEWAY 
1 
8080 : 8080 
labels : 
kompose . image- pull - secret : 
" regcred " 
kompose. service . expose: "{{ . Values . DNS NAME }}" 
kompose . service . type: "nodeport " 
kompose . service . nodeport . port • 
. "3eøø1" 

api-gateway service için uygulamanın görüleeği server bu olacağı için burada ekstra label lar var. daha sonra route53 te dns tanımlayacağımız için DNS label i da eklendi.

mysql -server : 
image: mysq1:5.7.8 
environment: 
MYSQL ROOT PASSWORD: petclinic 
MYSQL DATABASE: petclinic 
ports : 
3306 : 3306 

mysql server image i dockerhubdan çekilecek. password ve database verileri bize verilen proje docs ta var.

kompose convert -f docker-compose.yml -o petclinic\_chart/templates

bu komutla kompose tool unu çalıştırıyoruz. -f docker-compose.yml ise dosyanın ismi ve dosyanın yanında çalıştırırsak bu kısma gerek yok. -o ile de objeleri koymasını istediğimiz yeri berlirliyoruz.

v templates 
! admin-server-deployment.yaml 
! admin-server-service.yaml 
api-gateway-deploymentyaml 
api-gateway-ingress.yaml 
api-gateway-service.yaml 
! config-server-deploymentyaml 
! config-server-service.yaml 
customers-service-deployment.yaml 
customers-service-service.yaml 
discovery-server-deployment.yaml 
discovery-server-senice.yaml 
grafana-server-deployment.yaml 
grafana-server-service.yaml 
hystrix-dashboard-deploymentyaml 
hystrix-dashboard-service.yaml 
! k8s-default-networkpolicy.yaml 
mysql-server-deployment.yaml 
mysql-server-serv'ice.yaml 

templates içinde manifest dosyalarımızı oluşturdu kompose.

default bir networkpolicy de oluşturdu:

@ README.md 
k8s-default-networkpolicy.yaml X 
h-db > k8s > petclinic_chart > templates > ! k8s-default-networkpolicy.yal 
1 
2 
3 
5 
6 
7 
8 
le 
11 
12 
13 
14 
15 
apiVersion: networking. k8s . io/vl 
kind: NetworkP01icy 
metadata : 
creationTimestamp: null 
name: k8s-defau1t 
spec : 
Ingress : 
from : 
podSe1ector: 
matchLabe1s : 
io . kompose . network/k8s -defaul 
podSe1ector: 
matchLabe1s : 
io. kompose . network/k8s-defau1t: "true 

network policylerle podların birbiriyle iletişime girmesini confgüre ediyorduk.

**kompose convert -c -o petclinic-chart**

komutuyla da kompose chartı oluşturup dosyaları içine atar.

kompose tool u ile hazırladığımız dosyaları direk hazır olarak düşünmemeliyiz, eksik veya hatalı olabilir.

bizim serverlarımızın bir sırası vardı, önce config-serve sonra da discovery-server ayağa kalkamalı ve yayın yapmalıydı. bunu docker-compose da dockerize ile yapmıştık.

Kuberneteste ise busybox image i ile initContainer ile yapıyoruz.

bunun için deploymentlara bir düzeltme ekliyoruz.

Bir podun içinde birden fazla container olabiliyordu, initcontainer lar da birden fazla olabilir.

initContainers:

- name: init-config-server

image: busybox

command: ['sh', '-c', 'until nc -z config-server:8888; do echo waiting for config-server; sleep 2; done;']

bunu discovery-server deployment içine spec ltında container ile aynı hizaya yapıştırıyoruz.

initContainers:

- name: init-discovery-server

image: busybox

command: ['sh', '-c', 'until nc -z discovery-server:8761; do echo waiting for discovery-server; sleep 2; done;']

bunu da diğer serverlara (for admin-server, api-gateway, customers-service, hystrix-dashboard, vets-service and visits service) yapıştırıyoruz.

mı-2187 Fatih 
7:47 PM 
Init konteynerleri, bir pod'un başlatılma sürecinde çalışan ve pod'un ana 
konteynerlerinden önce çalıştırılan küçük özel konteynerlerdir. Init 
konteynerleri, pod'un başlatılması sırasında paralel olarak çalışır ve 
işlemlerini tamamladıktan sonra sona erer. 

['sh', '-c', 'until nc -z discovery-server:8761; do echo waiting for discovery-server; sleep 2; done;']

nc bir linux komutu, -z de nc komutunun parametresi dşnleme yap diyor. discovery-server:8761 podunu kontrol ettiriyoruz nc tool una. bu ismi service e verdiğimiz isimden alıyor.

buradan bilgi çekiyor, eğer bir şey alamazsa discovery serverı beliyorum diye yazıyor, 2 sn bekleyip tekrar dinliyor. dinleyip de yayın alıra done deyip çıkıyor. böyle bir komut.

api-gateway ingress.yaml a da bir ekleme yapacaz:

spec:

ingressClassName: nginx

rules:

- host: '{{ .Values.DNS\_NAME }}'

şimdi değil ancak EKS kullanırken ingress kullanacaz onun için şimdiden değiştiriyoruz. DNS.name kısmı da aslında var ancak küçük harfle geliyor (kompoe un bug ı) biz büyük harfe çeviriyoruz.

values-tmplate.yaml dosyamızı hazırlayacaz:

k8s/petclinic\_chart/values-template.yaml

envsub komutu ile de template i override yapacaz.

bu dosyanın içeriğine

IMAGE\_TAG\_CONFIG\_SERVER: "${IMAGE\_TAG\_CONFIG\_SERVER}"

IMAGE\_TAG\_DISCOVERY\_SERVER: "${IMAGE\_TAG\_DISCOVERY\_SERVER}"

IMAGE\_TAG\_CUSTOMERS\_SERVICE: "${IMAGE\_TAG\_CUSTOMERS\_SERVICE}"

IMAGE\_TAG\_VISITS\_SERVICE: "${IMAGE\_TAG\_VISITS\_SERVICE}"

IMAGE\_TAG\_VETS\_SERVICE: "${IMAGE\_TAG\_VETS\_SERVICE}"

IMAGE\_TAG\_API\_GATEWAY: "${IMAGE\_TAG\_API\_GATEWAY}"

IMAGE\_TAG\_ADMIN\_SERVER: "${IMAGE\_TAG\_ADMIN\_SERVER}"

IMAGE\_TAG\_HYSTRIX\_DASHBOARD: "${IMAGE\_TAG\_HYSTRIX\_DASHBOARD}"

IMAGE\_TAG\_GRAFANA\_SERVICE: "${IMAGE\_TAG\_GRAFANA\_SERVICE}"

IMAGE\_TAG\_PROMETHEUS\_SERVICE: "${IMAGE\_TAG\_PROMETHEUS\_SERVICE}"

DNS\_NAME: "petclinic.emreaydeniz.com"

yapıştırıyoruz. bu variable ları image leri tanımlamak için yazıyoruz.

DNS\_NAME kısmını da domain ismimize göre uygulamaya vereceğimiz ismi yazıyoruz. bunu daha sonra route53te ayarlayacaz.

şimdi de s3 bucket oluşturacaz ve helm repo haline getirecez. instance larımızda s3 e erişim yetkisi zaten vardı.

cli komutlarıyla yapacaz. aws s3 obje ekleme silme vs yapıyordu. bucket oluşturma komutu aws s3api ile oluyor.

aws s3api create-bucket --bucket petclinic-helm-charts-<put-your-name> --region us-east-1

aws s3api put-object --bucket petclinic-helm-charts-<put-your-name> --key stable/myapp/

bu komutlara kendi ismimizi veya unique olacak şekilde bir isim ekleyerek giriyoruz.

üstteki komut bucket oluştururken alttaki komut bucket içinde satble/myapp şeklinde prefix oluşturuyor.

helmi kurmamız direk s3 ü kllanabileceğimiz anlamına gelmşyor. s3 u kullanmak için helm-s3 plugini indirmemiz gerekiyor. helm artifact repo da bu plugini görebiliyoruz.

<https://artifacthub.io/packages/helm-plugin/s3/s3>

aws docsta da nasıl kullanacağımız anlatıyor:

<https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/set-up-a-helm-v3-chart-repository-in-amazon-s3.html>

helm plugin install <https://github.com/hypnoglow/helm-s3.git>

komutu ile plugini kuralım.

jenkins user ın bu plugini görebilmesi için jenkins-user ıa geçip bu plugini kurmamız gerekiyor.

export PATH=$PATH:/usr/local/bin

helm version

helm plugin install <https://github.com/hypnoglow/helm-s3.git>

önce helm in olduğu klasörü PATH e tanıttık. helm version ile bunu gördük ve helm plugini yükledik.

bir helm reponun olmazsa olmaz iki unsuru vardı, index.yaml ve chart.

init komutu ile index.yaml oluşturalım.

AWS\_REGION=us-east-1 helm s3 init s3://petclinic-helm-charts-<put-your-name>/stable/myapp

bucket ismini düzelterek bu komutu giriyoruz.

daha sonra da

aws s3 ls s3://petclinic-helm-charts-geovanni/stable/myapp/

komutuyla bucket içine bakıyoruz index.yam gelmiş mi diye.

ec2-user@jenkins-server# petclinic-microservices-with-db:( feature/msp-17 
• )$ aws s3 Is s3://petc1inic-he1m-charts-geovanni/stab1e/myapp/ 
2023-06-07 
2023-06-07 
71 index.yaml 

helm repo ls diyoruz herhangi bir repo yok.

ec2-user@jenkins-server# petclinic-microservices-with-db:( feature/msp-17 
@ )$ helm repo Is 
Error: no repositories to show 

AWS\_REGION=us-east-1 helm repo add stable-petclinicapp s3://petclinic-helm-charts-<put-your-name>/stable/myapp/

komutuyla da helm repo ekliyoruz. s3 bucketımızın url sini yazarak repoya bir isim veriyoruz bundan sona bu isimle bu repoya ulaşabilecez.

tekrar helm repo ls diyoruz.

ec2-user@jenkins-server# petclinic-microservices-with-db:( feature/msp-17 
• )$ helm repo Is 
stable-petclinicapp 
URL 
s3 : / / petclinic -helm-charts -geovanni/ stable/myapp/ 
ec2-user@jenkins-server# petclinic-microservices-with-db:( feature/msp-17 

repomuz geldi.

şimdi chart.yaml ı şu şekilde değiştiriyoruz:

version: 0.0.1

appVersion: 0.1.0

artık helm chart ı package yapabiliriz. package yapınca manifesto dosyalarımızı okaetleyecek ve install edebilecek şekilde zipleyecek. k8s altında helm package petclinic\_chart/

komutunu giriyoruz.

v k8s 
> petclinic_chart 
docker-compose.yml U 
petclinic_chart-O.O.1.tgz U 

buradaki 0.0.1 versionu chart.yaml dan geliyordu.

HELM\_S3\_MODE=3 AWS\_REGION=us-east-1 helm s3 push ./petclinic\_chart-0.0.1.tgz stable-petclinicapp

komutuyla da .tgz dosyamızı helm repoya push ediyoruz. helm pluginin docs unda v3 te bazen hata alınabilir diye HELM\_S3\_MODE=3 parametresini komutun başına ekleyin diye yazıyor. bu nedenle komuta eklendi bu kısım.

helm search repo stable-petclinicapp

ile de repoyu kontrol edebiliyoruz.

• ec2-user@jenkins-server# k8s:( feature/msp-17)$ helm search 
CHART VERSION APP VERSION 
stable-petclinicapp/petclinic_chart 
ø.ø.l 
ø.l.ø 
repo stable-petclinicapp 
DESCRIPTION 
A Helm chart for Kubernetes 

s3 ls ile de görebiliriz:

• ec2-user@jenkins-server# 
2023-06-07 
2023-06-07 
2023-06-07 
454 
3862 
feature/msp-17)$ aws s3 Is s3://petc1inic-he1m-charts-geovanni/stab1e/myapp/ 
index. yaml 
petclinic_chart -0.0.1. tgz 

chart.yal da versionu 0.0.2 yapıp tekrar paketliyoruz.

helm package petclinic\_chart/

şimdi de paketi tekrar gönderelim.

HELM\_S3\_MODE=3 AWS\_REGION=us-east-1 helm s3 push ./petclinic\_chart-0.0.2.tgz stable-petclinicapp

şimdi helm search repo stable-petclinicapp

komutuyla repoyu görelim. bazen görünmez ve update gerekir. helm repo update

komutyla önce update etmeyi alışkanlık haline getirelim. sonra helm search repo stable-petclinicapp

komutunu girelim.

• ec2-user@jenkins-server# k8s:( feature/msp-17)$ helm search 
CHART VERSION APP VERSION 
stable-petclinicapp/petclinic_chart 
ø.ø.2 
ø.l.ø 
repo stable-petclinicapp 
DESCRIPTION 
A Helm chart for Kubernetes 

helm search repo stable-petclinicapp --versions

komutuyla diğer versionları da görürüz.

Chart.yaml da version: karşısına HELM\_VERSION

yazıyoruz bunu daha sonra envsub komutu ile değiştirecez.

This is the chart version. 
to the chart and its templa 
Versions are expected to fo 
version: HELM VERSIO 

git add .

git commit -m 'added Configuration YAML Files for Kubernetes Deployment'

git push --set-upstream origin feature/msp-17

git checkout dev

git merge feature/msp-17

git push origin dev

1. adımı bitirdik.
2. adımdayız.
3. ееео 
   НЕИМ 
   clu%ter 
   ,qaster 

Şimdi MSP-18 de uygulamanın paketlenip, kuberneteste yağa kalkma sürecini jenkinsfile a aktaracğız.

git checkout dev

git branch feature/msp-18

git checkout feature/msp-18

Jenkins klasörü altında package-with-maven-container.sh

dosyası içine mvn clean package komutnu çalıştıracak bir docker komutu giriyoruz.

docker run --rm -v $HOME/.m2:/root/.m2 -v $WORKSPACE:/app -w /app maven:3.6-openjdk-11 mvn clean package

bu paketler bize lazım, dockerfile da .jar dosyasını çalıştıracak image oluşturmuştuk bu dosyalar mvn package komutuyla oluşacak.

bu komutun yapısını daha sonra da kullanmak üzere iyice öğrenmek gerekir.

şimdi yine jenkins klasörü altında prepare-tags-ecr-for-dev-docker-images.sh

isimli bir dosya altında docker image ler için ECR tag lerini oluşturacak komutlarını yazacaz..

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-admin-server/target/maven-archiver/pom.properties && echo $version)

export IMAGE\_TAG\_ADMIN\_SERVER="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:admin-server-v${MVN\_VERSION}-b${BUILD\_NUMBER}"

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-api-gateway/target/maven-archiver/pom.properties && echo $version)

export IMAGE\_TAG\_API\_GATEWAY="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:api-gateway-v${MVN\_VERSION}-b${BUILD\_NUMBER}"

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-config-server/target/maven-archiver/pom.properties && echo $version)

export IMAGE\_TAG\_CONFIG\_SERVER="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:config-server-v${MVN\_VERSION}-b${BUILD\_NUMBER}"

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-customers-service/target/maven-archiver/pom.properties && echo $version)

export IMAGE\_TAG\_CUSTOMERS\_SERVICE="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:customers-service-v${MVN\_VERSION}-b${BUILD\_NUMBER}"

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-discovery-server/target/maven-archiver/pom.properties && echo $version)

export IMAGE\_TAG\_DISCOVERY\_SERVER="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:discovery-server-v${MVN\_VERSION}-b${BUILD\_NUMBER}"

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-hystrix-dashboard/target/maven-archiver/pom.properties && echo $version)

export IMAGE\_TAG\_HYSTRIX\_DASHBOARD="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:hystrix-dashboard-v${MVN\_VERSION}-b${BUILD\_NUMBER}"

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-vets-service/target/maven-archiver/pom.properties && echo $version)

export IMAGE\_TAG\_VETS\_SERVICE="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:vets-service-v${MVN\_VERSION}-b${BUILD\_NUMBER}"

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-visits-service/target/maven-archiver/pom.properties && echo $version)

export IMAGE\_TAG\_VISITS\_SERVICE="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:visits-service-v${MVN\_VERSION}-b${BUILD\_NUMBER}"

export IMAGE\_TAG\_GRAFANA\_SERVICE="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:grafana-service"

export IMAGE\_TAG\_PROMETHEUS\_SERVICE="${ECR\_REGISTRY}/${APP\_REPO\_NAME}:prometheus-service"

biz buradaki variable ların isimlerini daha önce values-template.yaml da ve deploymentlarda image isimlerinde gördük.

işte bu variable ların tanımını burada oluşturuyoruz. buradaki amacımız dinamik bir tagleme yapmak. ileride güncellemelerden sonra hangi image in kullanıldığını bu sayede anlayabiliriz. örneğin iki ay sonra eski bir versiona dönmek istersek rahatça bulabilmemiz lazım bu nedenle tage latest vs demiyoruz da dinamik bir tagleme yapıyoruz.

${ECR\_REGISTRY}/${APP\_REPO\_NAME}

bu kısmı Jenkinsfile başında tanımlayacaz:

envlror—ent { 
App 
app 
AHS 
ECR 
_NAMF— 
_ RE P O_NJW E — " cla r.y - 
t S get-caller-identity 
_ dkr _ _ 
- -query Account - 
-output text' , 
returnStdout : true) . ) 
aNs_Ktypn1R- 
ANSI 
ANSIBLF I-OST CHECKING-"Fa1se" 

ECR\_REGISTRY ile APP\_REPO\_NAME i aws deki ECR repomuzdan alacak.

${BUILD\_NUMBER}

bu kısmı jenkinste tanımlı bir değişken ve hangi buildse o numarayı çekiyor.

${MVN\_VERSION}

developerlarçalıştıkça uyguamada değişiklik yaptıkça mvn versionu da değiştirebilirler bu nedenle bunu da dinamik olarak tage ekledik.

MVN\_VERSION=$(. ${WORKSPACE}/spring-petclinic-visits-service/target/maven-archiver/pom.properties && echo $version)

buradaki ilk nokta linux taki src komutunu ifade ediyor.

e c 2- us ere-jet* ins-server* 
ec2•use 
roe ins - server# 
2.1.2 
ec2- 
us e ins-server* 
test:$echo $version 
test:$_ test-txt 
test:$echo *version 
test.$0 

buradaki örnekte test.txt içindeki versionu echo ile yazdırmak için . test.txt ile bu dosyayı source olarak gösteriyoruz. şöyle de yazılır:

ec2-user@enkins - server* 
2.1.2 
ec2- 
u s e r&jeüins - serve" 
test:$source test . txt 
test: 
echo Sversion 

bu pipeline nightly açlışacak bu nedenle her seferinde verison veremeyiz. MVN\_VERSION u da dinamik şekilde çekmesini yukarıdaki komutla sağlamış olduk. bunu da tage eklemiş olduk.

Jenkinsfile tagleri tanımlayınca envsub komutuyla values-template.yaml ı values.yaml ın yerine geçirecez, override yapacaz. sonra deployment taki image de bu isimle image i çekecek.

jenkins klasörü altında build-dev-docker-images-for-ecr.sh

isimli dosya içine;

docker build --force-rm -t "${IMAGE\_TAG\_ADMIN\_SERVER}" "${WORKSPACE}/spring-petclinic-admin-server"

docker build --force-rm -t "${IMAGE\_TAG\_API\_GATEWAY}" "${WORKSPACE}/spring-petclinic-api-gateway"

docker build --force-rm -t "${IMAGE\_TAG\_CONFIG\_SERVER}" "${WORKSPACE}/spring-petclinic-config-server"

docker build --force-rm -t "${IMAGE\_TAG\_CUSTOMERS\_SERVICE}" "${WORKSPACE}/spring-petclinic-customers-service"

docker build --force-rm -t "${IMAGE\_TAG\_DISCOVERY\_SERVER}" "${WORKSPACE}/spring-petclinic-discovery-server"

docker build --force-rm -t "${IMAGE\_TAG\_HYSTRIX\_DASHBOARD}" "${WORKSPACE}/spring-petclinic-hystrix-dashboard"

docker build --force-rm -t "${IMAGE\_TAG\_VETS\_SERVICE}" "${WORKSPACE}/spring-petclinic-vets-service"

docker build --force-rm -t "${IMAGE\_TAG\_VISITS\_SERVICE}" "${WORKSPACE}/spring-petclinic-visits-service"

docker build --force-rm -t "${IMAGE\_TAG\_GRAFANA\_SERVICE}" "${WORKSPACE}/docker/grafana"

docker build --force-rm -t "${IMAGE\_TAG\_PROMETHEUS\_SERVICE}" "${WORKSPACE}/docker/prometheus"

docker build komutlarını giriyoruz.

bu komutun çözümlemesi:

docker build --force-rm -t "${IMAGE\_TAG\_ADMIN\_SERVER}" "${WORKSPACE}/spring-petclinic-admin-server"

-t ile tagleme yapıyoruz.

--force-rm de intermediate container ları isliyor.

Dockerfile da katmanlı yapı vardı:

petclinic-microsewices-with-db > spring-petclinic-admin-server > 
1 
2 
3 
5 
6 
7 
8 
le 
FROM openjdk:ll-jre 
ARG 
ARG 
ENV 
ADD 
RUN 
RUN 
ADD 
DOCKERIZE VERSION=vØ.6.1 
EXPOSED PORT-=gegø 
SPRING PROFILES ACTIVE docker,mysql 
https : //github.com/jwilder/dockerize/relea± 
tar -xzf dockerize.tar.gz 
chmod +x dockerize 
./target/* . jar /app.jar 
EXPOSE ${EXPOSED PORT} 
ENTRYPOINT ["java , ' 
" ' -Djava . security . egd=file: 

buradaki her aşamadan bir diğerine geçerken image alıyor ve her aşamada container lar ayağa kaldırıyor. biz --force-rm dersek arada takılı kalan container ları silmiş oluyoruz. CI/CD toollarında bu şekilde işlemler yapılır ve gereksiz şeyler silinir.

sıra image leri push etmeye geldi jenkins folder altında push-dev-docker-images-to-ecr.sh

isimli doysa içine

aws ecr get-login-password --region ${AWS\_REGION} | docker login --username AWS --password-stdin ${ECR\_REGISTRY}

docker push "${IMAGE\_TAG\_ADMIN\_SERVER}"

docker push "${IMAGE\_TAG\_API\_GATEWAY}"

docker push "${IMAGE\_TAG\_CONFIG\_SERVER}"

docker push "${IMAGE\_TAG\_CUSTOMERS\_SERVICE}"

docker push "${IMAGE\_TAG\_DISCOVERY\_SERVER}"

docker push "${IMAGE\_TAG\_HYSTRIX\_DASHBOARD}"

docker push "${IMAGE\_TAG\_VETS\_SERVICE}"

docker push "${IMAGE\_TAG\_VISITS\_SERVICE}"

docker push "${IMAGE\_TAG\_GRAFANA\_SERVICE}"

docker push "${IMAGE\_TAG\_PROMETHEUS\_SERVICE}"

push komutlarını yapıştırıyoruz.

önce ecr için login komutu sonra da push komutları. ecr a yetkisi olduğu için aws ecr get-login-password komutuyla password alabiliyor.

biz bu komutları jenkins pipeline ında .sh dosyaları halinde çalıştıracağız.

git add .

git commit -m 'added scripts for qa automation environment'

git push --set-upstream origin feature/msp-18

push layıp bgünlük burada bırakalım.