PROJECT TITLE

EMUMBA'S ASSIGNED PROJECT TO MUNEEB MUBASHIR FOR DEVOPS ENGINEER 1 POSITION

Although I was not having any advance knowledge of **kubernetes and GitHub actions** - yet I learned on my own and tried to complete the project in best possible manner.

LINK TO MY REPO: https://github.com/muneebxyz/cloudnative-EmumbaProject-Muneeb

LINK TO CLONE: https://github.com/muneebxyz/cloudnative-EmumbaProject-Muneeb.git

LINK TO APPLICATION: http://13.229.117.48:8080/

In case application is not loading use port forward command: ***kubectl port-forward --address 0.0.0.0 -n default service/frontend-service 8080:8080 &***

FILES:

- All files of this project including manifests for ***frontend***, ***api***, ***database***, ***network policy*** & ***CI github actions*** code are uploaded in the github repo (master branch) & are also attached with the email reply on which I got assignment.

APPROACH TO SOLVE PROBLEM

The approach that I am using to solve this assigned project is as follows explained in steps:

➢ FORKING, LOGGING IN, DOCKER BUILD & PUSH IMAGES

1- Firstly I have forked the given GITHUB repo to my own GITHUB account (muneebxyz).

2- After forking, I have logged in to the given VM in the email. As I was given username, password along with PublicIP & PrivateIP so I logged in using the following command:

ssh muneeb@PublicIP [After logging in entered the given Password]

3- Next I cloned the forked repo and my repo's clone link is: https://github.com/muneebxyz/cloudnative-EmumbaProject-Muneeb.git

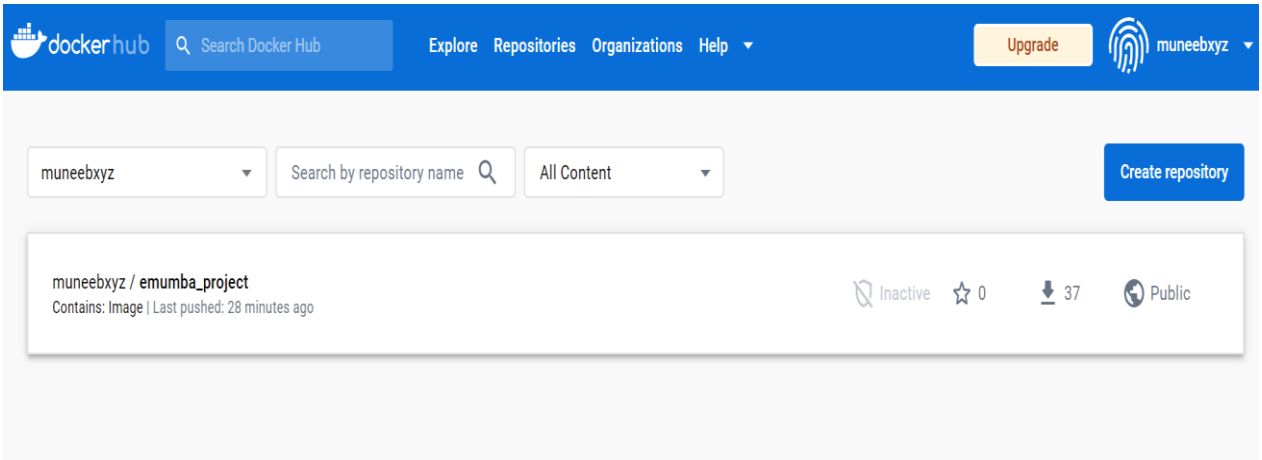
4- After cloning it, I headed to the **client/** directory where there is Dockerfile for frontend (ReactJS). I build the image from Dockerfile using:

docker build -t client_image .

5- Than I headed to the server/ directory where Dockerfile of backend is saved and I built the image using:

docker build -t server_image .

6- Next was the step to upload the built files (client_image & server_image) to DockerHub. For uploading to DockerHub I used the docker tag command so that I can configure the DockerHub account & repository inside it with the images. **The repo name of DockerHub is: emumba_project and is a public repo**



docker tag client_image muneebxyz/emumba_project:frontend

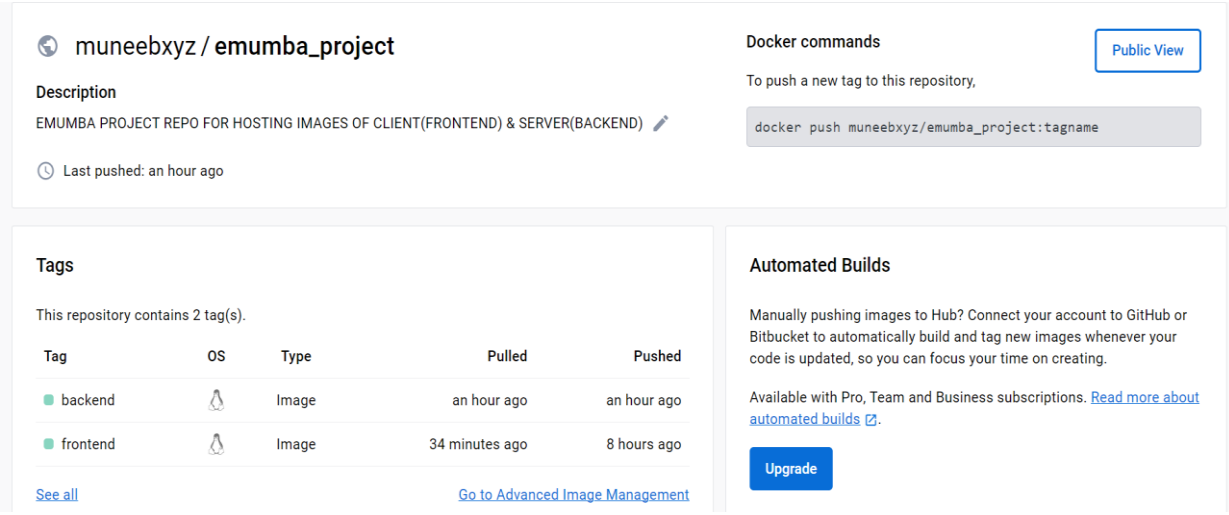
docker tag server_image muneebxyz/emumba_project:backend

7- Next I logged in to dockerhub using ***docker login*** command

8- Finally I pushed the images to dockerHub with the updated tags by using following commands:

docker push muneebxyz/emumba_project:frontend

docker push muneebxyz/emumba_project:backend



IMPLEMENTING CI USING GITHUB ACTIONS

1- The CI (Continuous Integration) pipeline is implemented by me by heading towards the actions tab in repo and than creating my own workflow in main.yaml file

[LOCATION: cloudnative-EmumbaProjectMuneeb/.github/workflows/main.yml]

APPROACH OF THE BUILD PIPELINE: It was assigned that build must trigger when any change in client directory is done and similarly if any change in server directory is done than also build should trigger.

➞ The main.yaml file consists of two jobs frontend & backend that will get trigger when we push to the master branch [If we make any change to the client or server directories]

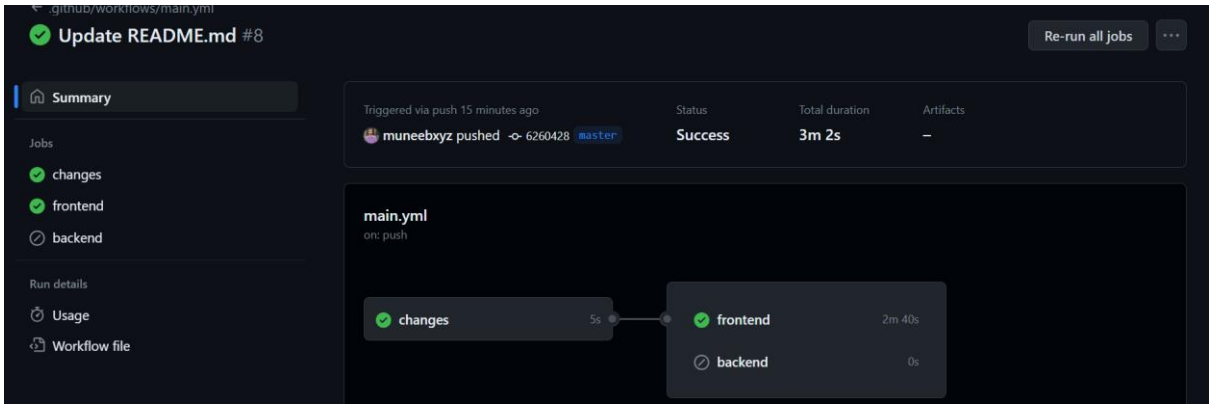
➞ When the change is detected in either directory using dorny/paths filter & corresponding if statements for detecting changes in both jobs of frontend and backend than the build process starts.

➞ In the pipeline, it is configured to first ***cd*** to the respective client or server directory depending on trigger and than ***BUILD, LOGIN TO DOCKERHUB ACCOUNT & PUSH UPDATED IMAGES TO DOCKERHUB***.

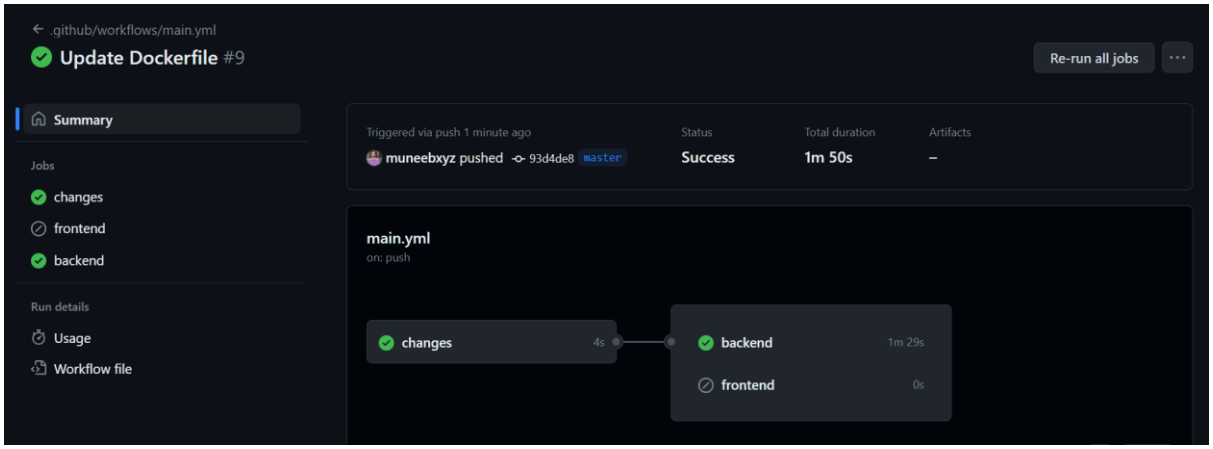
One Step more Further Implementation:

I have implemented a mechanism that when the change in client directory is taken place than only build for client should run and backend should be skipped and viceversa. The build first catches/detects the changes. And I implemented it using filters.

FRONTEND BUILD WHEN CLIENT DIRECTORY'S README FILE IS EDITED



BACKEND BUILD WHEN SERVER DIRECTORY'S Dockerfile IS EDITED



MINIKUBE AND KUBERNETES DEPLOYMENT

1- For deploying application, I first installed minikube by consulting its documentation. I installed and configured minikube using docker as a driver.

2- Than I started a single node cluster of minikube using the command:

minikube start

minikube status [To check status of cluster]

```
muneeb@ip-10-0-16-88:~$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

3- To get the information about running nodes we can use:

kubectl get nodes

```
muneeb@ip-10-0-16-88:~$ kubectl get nodes
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   12h   v1.26.3
```

4- Next is the time to start configuring the manifests for frontend, backend & db which I have also uploaded to the same GIT REPO that I forked.

➞ Next I created total 6 manifests (YAML) files.

➞ 2 files for frontend (frontend-service.yaml & frontend-deployment.yaml)

➞ 2 files for api (api-service.yaml & api-deployment.yaml)

➞ 2 files for database (db-service.yaml & db-deployment.yaml)

➞ I have linked the service and deployment manifests by ***selectors*** that are of same value

FRONTEND MANIFESTS:

- In frontend-service I have simply created a service and configured its port & nodeport.
- In frontend-deployment, I have set its ***replicas=1*** as kubernetes create the pods from replicas. Than I have established the containers using image in my dockerhub public repo as: ***muneebxyz/emumba_project:frontend***

DATABASE MANIFESTS:

- In the db-service file, I have created its service and configured its ports as its ***mongodb*** so its port is confgigured to 27017.
- In the db-deployment file, I have created its pod using the same replicas=1 and than container is created using the ***bitnami/mongodb:latest** as image

➞ **One of the thing that I seeked help from docker-compose.yaml** file was the environment variables for the database. Which is configured in db-deployment.yaml file as: MONOGODB_ROOT_PASSWORD, MONGODB_DATABASE, MONGODB_USERNAME, MONGODB_PASSWORD.

```
env:
    - name: MONGODB_ROOT_PASSWORD
      value: admin
    - name: MONGODB_DATABASE
      value: admin
    - name: MONGODB_USERNAME
      value: admin
    - name: MONGODB_PASSWORD
      value: admin
```

API [SERVER] MANIFESTS:

➞ In the api-service.yaml file, I have created the api service and configured its ports.

➞ For ports I have used **type:ClusterIP** which means that this service is accessible from only within the cluster and it exposes the port on internal communication. And its targetport is configured to 8080 on which it (pod) shall listen.

➞ For the api-deployment.yaml file, I have created the same replicas=1 and created the container from the image on my dockerhub repo as: ***muneebxyz/emumba_project:backend***

➞ API container depends on db service so I have to configure its environment variables as:

env:

- name: DB_CONNECTION

value: mongodb://admin:admin@db-service:27017/admin

PODS STATUS CHECK & OTHER COMMANDS:

After creating all the manifests files and linking their services with their deployments file now is the time to apply them to establish the pods out of them.

COMMAND: kubectl apply -f <manifest-file-name.yaml>

In this way I applied all the 6 files.

➞ The pod status can be viewed from command:

kubectl get pods

```
muneeb@ip-10-0-16-88:~$ kubectl get pods
NAME                                  READY   STATUS    RESTA
RTS   AGE
api-deployment-77c96468fc-csf8b       1/1     Running   0
      12h
db-deployment-889b98f96-mpnzq         1/1     Running   0
      12h
frontend-deployment-555499c7c7-pv47f  1/1     Running   0
      12h
muneeb@ip-10-0-16-88:~$
```

All pods are in healthy running condition and have no errors.

➞ The total running services can be viewed from:

kubectl get svc

```
muneeb@ip-10-0-16-88:~$ kubectl get svc
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)           AGE
api-service       ClusterIP   10.98.215.4      <none>        8080/TCP          12h
db-service        NodePort    10.100.165.221   <none>        27017:31859/TCP   12h
frontend-service  NodePort    10.106.160.102   <none>        8080:30000/TCP    12h
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP           13h
```

➞ The total deployments on cluster can be viewed from:

kubectl get deploy

```
muneeb@ip-10-0-16-88:~$ kubectl get deploy
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
api-deployment       1/1     1            1           12h
db-deployment        1/1     1            1           12h
frontend-deployment  1/1     1            1           12h
muneeb@ip-10-0-16-88:~$
```

➞ Finally the status of all services along with their IPs can be viewed from:

minikube service list

```
muneeb@ip-10-0-16-88:~$ minikube service list
|-------------|-----------------|--------------|---------------------------|
|  NAMESPACE  |       NAME      | TARGET PORT  |            URL            |
|-------------|-----------------|--------------|---------------------------|
| default     | api-service     | No node port |                           |
| default     | db-service      |        27017 | http://192.168.49.2:31859 |
| default     | frontend-service |         8080 | http://192.168.49.2:30000 |
| default     | kubernetes      | No node port |                           |
| kube-system | kube-dns        | No node port |                           |
|-------------|-----------------|--------------|---------------------------|
```

BONUS SECTION :

In this it was asked to restrict the traffic between db & frontend pods.

➞ I have implemented it by consulting it from documentation and using the **ingress** rule & in which the only traffic is allowed from frontend and backend and only they can communicate internally and no traffic from external means can interfere.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-communication-frontend-db
spec:
  podSelector:
    matchLabels:
      app: frontend
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: db
```

The code is simple and using the apiVersion of networking of kubernetes to implement the network policies. In the spec, the pod selector is defined which matches the app name and than ingress is used to redirect the traffic between only frontend & db.