

## Chapter 3: Internship Activities

### 3.1. Roles and Responsibilities

While working as a Backend Python Developer intern for Khalti, my main focus was in the development and refinement of a various feature, which involved designing and implementing backend solutions using Python and Django, ensuring these systems were robust, scalable, and seamlessly integrated with the existing digital wallet platform. I had the following tasks:

- i) **Code Development and Testing:**  
I developed new backend features and enhanced existing functionalities using Python and Django, and I conducted thorough testing to ensure robustness and reliability.
- ii) **System Maintenance and Optimization:**  
I carried out routine maintenance and optimization tasks to improve the efficiency and scalability of the backend systems.
- iii) **Database Management:**  
I managed database schemas and integrated data storage solutions, ensuring data consistency and security.
- iv) **Implementation of Security Protocols:**  
I implemented and reviewed security measures to safeguard the digital wallet against potential threats and vulnerabilities.
- v) **Collaboration with Other Teams:**  
I maintained close collaboration with the front-end development team and other technical departments to ensure seamless integration and functionality across platforms.
- vi) **Documentation of Development Processes:**  
I created and maintained comprehensive documentation of all development processes, changes, and upgrades, providing a clear reference for future development efforts and troubleshooting.

### 3.2. Weekly log

Following table shows the weekly activities the intern performed throughout their internship period.

**Table 3.1 : Weekly Log**

Week	Activities
Week 1	<ul style="list-style-type: none"><li>• Orientation</li><li>• Installed CentOS with automatic as well as manual partition and got familiar with Linux commands</li><li>• Learned hosting HTML, CSS on nginx and tomcat webserver</li><li>• Setup nginx for reverse proxy and load balancing</li></ul>
Week 2	<ul style="list-style-type: none"><li>• Explored about HAProxy for load balancing</li><li>• Configured firewalld in CentOS for allowing traffic</li><li>• Got familiar with SELinux policies</li></ul>

	<ul style="list-style-type: none"> <li>• Hosted tomcat website with Dockerfile</li> <li>• Learned basic commands used in redis for retrieving and inserting data</li> <li>• Configured keepalived for loadbalancing and high-availability</li> <li>• Configured self hosted git server without GUI</li> <li>• Configured nginx hosted website with SSL certificate</li> </ul>
Week 3	<ul style="list-style-type: none"> <li>• Learned to create own Certificate Authority and generate SSL certificate for client</li> <li>• Setup mysql database</li> <li>• Installed and configured MinIO for managing unstructured object data</li> <li>• Installed monit for process monitoring and configured to send alert email</li> <li>• Hosted git service like github, gitlab</li> <li>• Configured uptime-kuma for monitoring services over HTTP/S, TCP, DNS and other protocols</li> <li>• Learned to create bash script for send email using curl</li> <li>• Achieved file synchronization between master with worker nodes using lsyncd</li> <li>• Learned about Network File System for sharing files between nodes within a network</li> <li>• Learned about traefik for reverse proxy and load balancing</li> </ul>
Week 4	<ul style="list-style-type: none"> <li>• Configured MySQL replication for data backup</li> <li>• Setup self-hosted registry for docker images i.e. Harbor</li> <li>• Configured jenkins to push to the self-hosted registry</li> <li>• Learned to create ansible playbooks</li> <li>• Configured grafana, loki and prometheus for log collection and their visualization</li> <li>• Studies about cryptography and PKI</li> <li>• Setup k3s with rancher to manage kubernetes cluster</li> </ul>
Week 5	<ul style="list-style-type: none"> <li>• Deploy website in nginx using k3s pod</li> <li>• Learned about kubernetes concepts</li> <li>• Deployed sidecar container in k3s</li> <li>• Learned to manage LVM partition of linux system</li> <li>• Deployed a website synchronized with git repo</li> <li>• Learned about awk and sed command</li> </ul>
Week 6	<ul style="list-style-type: none"> <li>• Configured k3s deployed website with SSL certificate using k3s TLS Secrets</li> <li>• Learned about ArgoCD for continuous delivery</li> </ul>

### 3.3. Description of the Project(s) Involved During Internship

One of the highlights which really helped me in understanding DevOps principles was working on two minor projects during my internship both using local infrastructure.

## Project 1: Setting up Jenkins and Self-Hosted Docker Registry

In this project, my main goal was to automate the build and deployment processes by using Jenkins as well as a self-hosted docker registry.

- i) **Under Jenkins Configuration:** I installed Jenkins on a local server then did its setup for managing CI/CD pipelines which were continuous integration and continuous deployment. This involved creating Jenkins jobs that would automate building and testing phases of application development.
- ii) **Setting Up Self-Hosted Docker Registry:** I established an internal docker registry within a server environment so that we could host our docker images locally instead of relying on external registries where we have less control over them or their security levels.
- iii) **Image Push To Registry:** Thereafter, I configured jenkins pipelines; these would pull source codes from different repositories then build corresponding docker images before pushing these artifacts into my own registry which I had set up before hand thus ensuring all necessary files for deployment are stored in one place for both security

The result was a complete automation of CI/CD pipeline which eliminated manual intervention, reduced errors and made deployment faster within our premises.

## Project 2: Using git-sync with Nginx Deployed In K3S

During this project work centered around syncing websites hosted on self hosted git services (Gitea/ Gogs) with nginx servers deployed inside K3s clusters as well adding SSL certificates deployment along side Mysql replication also .

- i) **Git-sync Implementation:** Configured git-sync within the local K3s environment to automatically pull the latest changes from the Git repository (hosted on a local Gitea or Gogs instance). This ensured that the website content was always up-to-date with the latest code changes.
- ii) **Nginx Deployment:** Deployed Nginx as a reverse proxy in the local K3s cluster to serve the website. Configured Nginx to serve the synchronized content from the Git repository.
- iii) **SSL Certificates:** Utilized Kubernetes Ingress with TLS secrets to implement SSL certificates for secure HTTPS access to the website. This was done using Let's Encrypt to automatically issue and renew certificates.
- iv) **Monitoring and Logging:** Implemented monitoring and logging solutions using Grafana, Prometheus, and Loki. These tools were configured to monitor the performance and health of the applications, collect logs, and visualize metrics, enabling proactive management and troubleshooting of the deployed applications.

The project demonstrated a robust, automated deployment process with real-time synchronization, secure access, and reliable data replication, all managed within the local infrastructure.

These projects provided hands-on experience with several key DevOps tools and practices, such as Jenkins, Docker, K3s, git-sync, Nginx, SSL certificates. The practical knowledge gained from these projects significantly enhanced my skills in automating, managing, and securing deployments in a production environment, all within the scope of local infrastructure.

### 3.4. Tasks / Activities Performed

#### i) Pushing docker image to self hosted container registry

In this task, Docker images have been pushed to self hosted container registry using self hosted jenkins by running shell script. Given steps were being followed to do the task:

- Install Jenkins and it's dependencies using following shell script:

**File: Jenkins-install.sh**

```
sudo wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/redhat/jenkins.repo
sudo rpm --import https://pkg.jenkins.io/redhat/jenkins.io-2023.key
sudo dnf upgrade
# Add required dependencies for the jenkins package
sudo dnf install fontconfig java-17-openjdk
sudo dnf install jenkins
sudo systemctl enable jenkins
sudo systemctl start jenkins

JENKINS_PORT=8080
PERM="--permanent"
SERV="$PERM --service=jenkins"

firewall-cmd $PERM --new-service=jenkins
firewall-cmd $SERV --set-short="Jenkins ports"
```

```
firewall-cmd $SERV --set-description="Jenkins port exceptions"
firewall-cmd $SERV --add-port=$JENKINS_PORT/tcp
firewall-cmd $PERM --add-service=jenkins
firewall-cmd --zone=public --add-service=http --permanent
firewall-cmd --reload
```

- Install and configure Harbor which is self hosted registry using docker with following script and create a new project from Harbor Dashboard.

**File: Harbor-install-and-configure.sh**

```

mkdir -p harbor-setup; cd harbor-setup
wget https://github.com/goharbor/harbor/releases/download/v2.11.0/harbor-online-
installer-v2.11.0.tgz
tar xvf harbor-online-installer-v2.11.0.tgz
openssl genrsa -out ca.key 4096

echo -e "\n-----\n Creating Root CA Certificate \n-----\n "
openssl req -x509 -new -nodes -sha512 -days 3650 \
  -subj "/C=NP/ST=Bagmati Pradesh/L=Kathmandu/O=MeroOrganization/OU=Personal/
CN=merodomain.com" \
  -key ca.key \
  -out ca.crt

echo -e "\n-----\n Generating Server Certificate \n-----\n"
openssl genrsa -out meroharbor.com.key 4096
openssl req -sha512 -new \
  -subj "/C=NP/ST=Mero State/L=Mero Thau/O=Mero Organization/OU=Personal/
CN=meroharbor.com" \
  -key meroharbor.com.key \
  -out meroharbor.com.csr

cat > meroharbor.com.v3.ext <<-EOF
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names
[alt_names]
DNS.1=meroharbor.com
EOF

echo -e "\n-----\n Signing Server Certificate with CA's Certificate
\n-----\n"

```

```

openssl x509 -req -sha512 -days 3650 \
  -extfile meroharbor.com.v3.ext \
  -CA ca.crt -CAkey ca.key -CAcreateserial \
  -in meroharbor.com.csr \
  -out meroharbor.com.crt

echo -e "\n-----\n Converting crt to cert \n-----\n"
openssl x509 -inform PEM -in meroharbor.com.crt -out meroharbor.com.cert
mkdir -p /etc/docker/certs.d/meroharbor.com:443/
cp meroharbor.com.cert /etc/docker/certs.d/meroharbor.com:443/
cp meroharbor.com.key /etc/docker/certs.d/meroharbor.com:443/
cp ca.crt /etc/docker/certs.d/meroharbor.com:443/
cd harbor; cp harbor.yml.tpl harbor.yml
sed -i 's|hostname: reg.mydomain.com|hostname: meroharbor.com|; s|certificate: /your/
certificate/path|certificate: /etc/docker/certs.d/meroharbor.com:443/
meroharbor.com.cert|; s|private_key: /your/private/key/path|private_key: /etc/docker/
certs.d/meroharbor.com:443/meroharbor.com.key|' harbor.yml
./prepare
docker compose up -d

```

Following picture provide the output of the above shell script:

```
root@college-report ~/h/harbor# docker compose up -d
WARN[0000] /root/harbor-setup/harbor/docker-compose.yml: 'version' is obsolete
[+] Running 9/0
 ✓ Container harbor-log      Running      0.0s
 ✓ Container redis           Running      0.0s
 ✓ Container registry        Running      0.0s
 ✓ Container harbor-portal    Running      0.0s
 ✓ Container registryctl     Running      0.0s
 ✓ Container harbor-db        Running      0.0s
 ✓ Container harbor-core      Running      0.0s
 ✓ Container nginx            Running      0.0s
 ✓ Container harbor-jobservice Running      0.0s
root@college-report ~/h/harbor#
```

Figure 3.1 : Harbor Installation

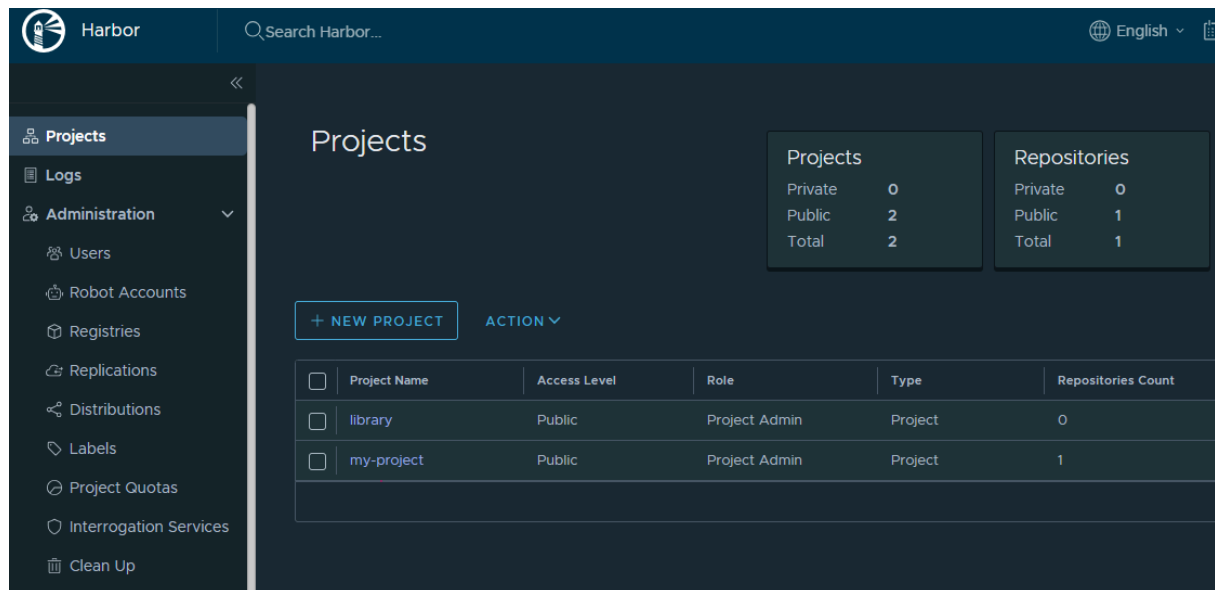


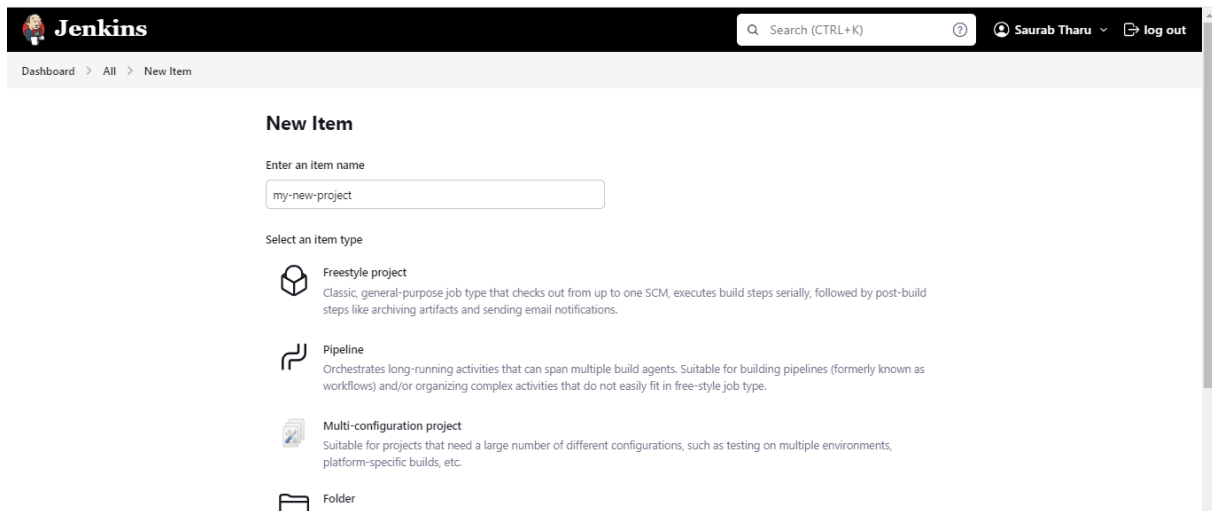
Figure 3.2 : Harbor Project Creation

- Create Dockerfile to build docker image

**File: Dockefile**

```
FROM nginx:latest
COPY ./mysite /usr/share/nginx/html/
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

**Step 4:** Create a Freestyle Project in Jenkins



**Figure 3.3 : Creating Jenkins Freestyle Project**

- Configure the Jenkins project by adding **“Execute shell”** build step and use the following shell script and save the configuration.

```
DOCKER_REGISTRY='meroharbor.com:443'
DOCKER_REPO='my-project'
IMAGE_NAME='my_web'
TAG="build-${BUILD_NUMBER}"


cd /var/lib/jenkins/workspace/MyWeb-DockerFile/
ls -al

docker build -t ${DOCKER_REGISTRY}/${DOCKER_REPO}/${IMAGE_NAME}:${TAG} .

# here password to login is stored in some.txt file
cat some.txt | docker login ${DOCKER_REGISTRY} -u saurab --password-stdin

docker push ${DOCKER_REGISTRY}/${DOCKER_REPO}/${IMAGE_NAME}:${TAG}
docker logout
```

- Finally build the project and check the **“Console Output”** to verify if build was successful or not.

 **Jenkins**

Search (CTRL+K) ?

Dashboard > push to my harbor > #61 > Console Output

Status

Changes

**Console Output**

Edit Build Information

Delete build '#61'

Timings

Previous Build

✓ Console Output

Download Copy View as plain text

Started by user [Saurab Tharu](#)

Running as SYSTEM

Building in workspace /var/lib/jenkins/workspace/push to my harbor

[push to my harbor] \$ /bin/sh -xe /tmp/jenkins2951868309193692232.sh

+ DOCKER\_REGISTRY=meroharbor.com:443

+ DOCKER\_REPO=my-project

+ IMAGE\_NAME=my\_web

+ TAG=build-61

+ cd /var/lib/jenkins/workspace/MyWeb-DockerFile/

+ ls -la

total 8

drwxr-xr-x 3 jenkins jenkins 54 Jun 11 22:54 .

drwxr-xr-x 4 jenkins jenkins 55 Jun 11 20:17 ..

-rw-r--r-- 1 jenkins jenkins 101 Jun 11 20:17 Dockerfile

drwxr-xr-x 7 jenkins jenkins 81 Jun 11 20:17 mysite

-rw-r--r-- 1 root root 13 Jun 11 22:54 some.txt

+ pwd

/var/lib/jenkins/workspace/MyWeb-DockerFile

+ docker build -t meroharbor.com:443/my-project/my\_web:build-61 .

#0 building with "default" instance using docker driver

Figure 3.4 : Jenkins Console Output



## ii) Dockerizing nginx hosted website and providing SSL certificate to the website

Hosting a website using Nginx and securing it with an SSL certificate is one of the fundamental task DevOps needs to understand. This involved preparing a Dockerfile and configuring the Nginx server to serve the website content and setting up virtual hosts. To ensure secure communication, SSL certificates were generated and installed. openssl tool is used for generating certificates which includes certificate.crt file and private key .key file. Following files were used for dockerization of nginx website:

### File: generate\_certificate.sh

```
#!/bin/bash

set -x
openssl genrsa -out ca.key 4096

echo -e "\n-----\n Creating Root CA Certificate \n-----\n "
openssl req -x509 -new -nodes -sha512 -days 3650 \
  -subj "/C=NP/ST=Bagmati Pradesh/L=Kathmandu/O=MeroOrganization/OU=Personal/
CN=meroCA.com" \
  -key ca.key \
  -out ca.crt

echo -e "\n-----\n Generating Server Certificate \n-----\n"
openssl genrsa -out merowebste.com.key 4096
openssl req -sha512 -new \
  -subj "/C=NP/ST=Mero State/L=Mero Thau/O=Mero Organization/OU=Personal/
CN=merowebste.com" \
  -key merowebste.com.key \
  -out merowebste.com.csr

cat > merowebste.com.v3.ext <<-EOF
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[alt_names]
DNS.1=merowebste.com
EOF
```

```
echo -e "\n-----\n Signing Server Certificate with CA's Certificate
\n-----\n"
openssl x509 -req -sha512 -days 3650 \
  -extfile merowebste.com.v3.ext \
  -CA ca.crt -CAkey ca.key -CAcreateserial \
  -in merowebste.com.csr \
  -out merowebste.com.crt
```

### File: nginx.conf

```
server {  
    listen      443;  
    server_name merowebste.com;  
  
    ssl on;  
    ssl_certificate      /usr/share/nginx/certificates/merowebste.com.crt;  
    ssl_certificate_key  /usr/share/nginx/certificates/merowebste.com.key;  
  
    location / {  
        root    /usr/share/nginx/html;  
        index   index.html index.htm;  
    }  
}
```

**File: Dockerfile**

```
FROM nginx:1.10.1-alpine  
COPY jd /usr/share/nginx/html  
COPY merowebste.com.crt /usr/share/nginx/certificates/  
COPY merowebste.com.key /usr/share/nginx/certificates/  
COPY nginx.conf /etc/nginx/conf.d/default.conf  
EXPOSE 443  
CMD ["nginx", "-g", "daemon off;"]
```

### iii) Configuring up load balancing

Suppose we have four website hosted using nginx at port 1111, 2222, 3333, 4444 with following nginx configuration

**File: website-for-load-balancing.conf**

```
#####  
#   This is for loadbalancing  
#####  
  
# localhost:1111  
server {  
    listen      1111;  
    server_name localhost;  
    root        /usr/share/nginx/digital-trend;  
    index index.html index.htm  
}  
  
# localhost:2222  
server {  
    listen      2222;  
    server_name localhost;  
    root        /usr/share/nginx/jackpiro;  
    index index.html index.htm  
}  
  
# localhost:3333  
server {  
    listen      3333;  
    server_name localhost;  
    root        /usr/share/nginx/memorial;  
    index index.html index.htm  
}  
  
# localhost:4444  
server {  
    listen      4444;  
    server_name localhost;  
    root        /usr/share/nginx/shiphile;  
    index index.html index.htm  
}
```

### a) Using Nginx

The following Nginx configuration can be used to set up a load balancer that distributes traffic across multiple backend servers. This setup ensures that requests are balanced between the servers specified in the upstream block.

**File: haproxy.cfg**

```
upstream myapp1 {
    server localhost:1111;
    server localhost:2222;
    server localhost:3333;
    server localhost:4444;
}
server {
    listen 80;
    server_name nginx-load-balancer.com;           # domain-name
    location / {
        proxy_pass http://myapp1;
    }
}
```

### b) Using HAProxy

The following configuration can be used for HAProxy to load balance traffic across multiple nodes, ensuring that if any node goes down, the other nodes will continue to serve the website. This setup distributes incoming traffic using the round-robin algorithm and checks the health of each node to maintain high availability.

**File: haproxy.cfg**

```
frontend main
    bind *:80
    timeout client 60s
    mode http
    default_backend backend_name # <- name of backend specified below
backend backend_name
    timeout connect 30s
    timeout server 100s
    mode http
    balance roundrobin
    http-request set-header Host load-balanced-website.com
    server node1 127.0.0.1:1111 check
    server node2 127.0.0.1:2222 check
    server node3 127.0.0.1:3333 check
    server node4 127.0.0.1:4444 check
```

## v) Configuring k3s for synchronizing deployed website with code in Gitea

K3s, a lightweight Kubernetes distribution, was used for managing containerized applications efficiently. Synchronizing a deployed website with a GitHub repository ensures continuous integration and deployment, allowing the website to reflect the latest code changes automatically. At first setting up the environment involves installing k3s, a lightweight Kubernetes distribution that simplifies cluster setup and management. By running a single command, we install k3s and initiate the server. Configuring kubectl ensures that we can manage the k3s cluster efficiently, setting the stage for deploying applications. This streamlined setup process demonstrates k3s's ease of use and quick deployment capabilities.

To install k3s, execute the following command in your terminal. This command downloads and installs the k3s binary and starts the k3s server.

```
$ curl -sL https://get.k3s.io | sh -
```

Deploying a sample website on k3s involves creating a deployment configuration file (deployment.yaml) that defines the desired state of the application, including the number of replicas, container images, and service specifications. Applying this configuration with kubectl deploys the website on the k3s cluster.

**File: nginx-deploymentgit.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy-resturan
  namespace: nginx
  labels:
    app: nginx-cc
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-cc
  template:
    metadata:
      labels:
        app: nginx-cc
    spec:
      containers:
        - name: git-sync
          image: registry.k8s.io/git-sync/git-sync:v4.2.3
```

```

    volumeMounts:
      - name: www-data
        mountPath: /data
    env:
      - name: GITSYNC_REPO
        value: "http://192.168.33.11:3000/saurab/website-for-git-sync.git"
      - name: GIT_SYNC_BRANCH
        value: "main"
      - name: GITSYNC_ROOT
        value: /data
      - name: GIT_SYNC_DEST
        value: "html"
      - name: GITSYNC_ONE_TIME
        value: "false"
      - name: GIT_SYNC_COMMAND_AFTER
        value: "nginx -s reload"
    securityContext:
      runAsUser: 0
  - name: nginx-resturan
    image: nginx
    ports:
      - containerPort: 80
    volumeMounts:
      - mountPath: "/usr/share/nginx/"
        name: www-data
  volumes:
    - name: www-data
      emptyDir: {}

```

This nginx-ingress.yaml file defines an Ingress resource named nginx-ingress using the Traefik controller, with TLS termination via tls-secret for roltu.com. It routes all HTTP requests to roltu.com to the nginx-service on port 80.

**File: ingnx-ingress.yaml**

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
  annotations:
    kubernetes.io/ingress.class: "traefik"

```

```

spec:
  tls:
    - secretName: tls-secret
      hosts:
        - "roltu.com"
  rules:
    - host: roltu.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: nginx-service
                port:
                  number: 80

```

This nginx-service.yaml defines a Kubernetes Service named nginx-service in the nginx namespace. It uses a NodePort to expose port 80, forwarding traffic from the service on port 80 to the pods labeled app: nginx-cc on port 80, accessible via node IP on port 30002.

**File: nginx-namespace.yaml**

```
apiVersion: v1
kind: Namespace
metadata:
  name: nginx
```

**File: nginx-service.yaml**

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: nginx
spec:
  type: NodePort
  selector:
    app: nginx-cc
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30002
```

Apply the configuration to the k3s cluster using kubectl using commands as below:

```
$ kubectl apply -f nginx-namespace.yaml
$ kubectl apply -f nginx-deploymentgit.yaml
$ kubectl apply -f nginx-ingress.yaml
$ kubectl apply -f nginx-service.yaml
```

The above deployment file to work we need a git repo so for that self hosted Gitea was used. To install the Gitea following bash script is used:

**File: gitea-installation.sh**

```
#!/bin/bash
# Create a directory and navigate into it
mkdir gitea
cd gitea
wget -O gitea https://dl.gitea.com/gitea/1.22.0/gitea-1.22.0-linux-amd64
chmod +x gitea
useradd git
groupadd git
groupadd git
chown -R root:git gitea
chmod g+s gitea
mkdir -p /usr/local/bin/gitea
mv gitea /usr/local/bin/gitea

cat > /etc/systemd/system/gitea.service <<-EOF
[Unit]
Description=Gitea (Git with a cup of tea)
After=syslog.target
After=network.target
#After=mysql.service
#After=postgresql.service
#After=memcached.service
#After=redis.service

[Service]
# Modify these two values and uncomment them if you have
# repos with lots of files and get an HTTP error 500 because
# of that
#LimitMEMLOCK=infinity
#LimitNOFILE=65535
RestartSec=2s
Type=simple
```

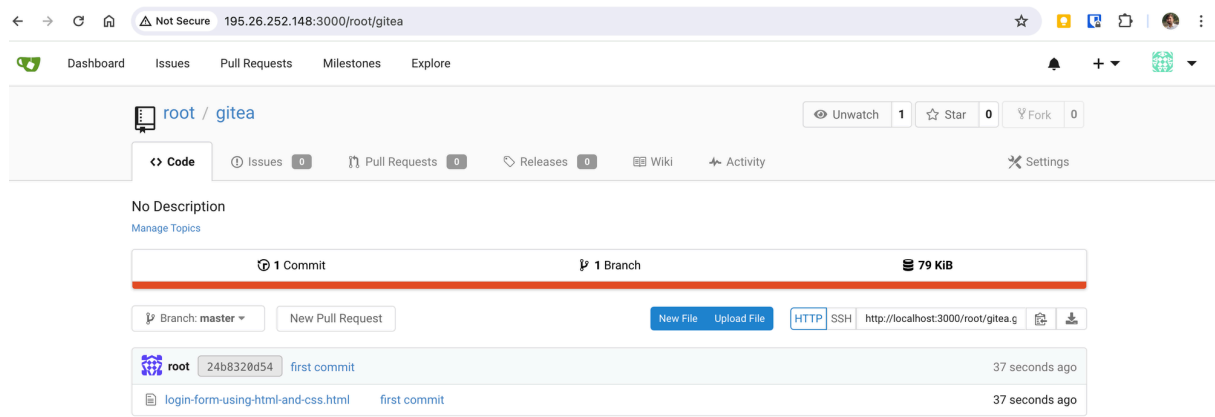
```
User=git
Group=git
WorkingDirectory=/usr/local/bin/gitea
ExecStart=/usr/local/bin/gitea/gitea web
Restart=always
Environment=USER=git HOME=/home/git
# If you want to bind Gitea to a port below 1024 uncomment
# the two values below
#CapabilityBoundingSet=CAP_NET_BIND_SERVICE
#AmbientCapabilities=CAP_NET_BIND_SERVICE

[Install]
WantedBy=multi-user.target
EOF

systemctl start gitea.service
systemctl enable gitea.service
```

After installation, the gitea can be accessed using ipaddress along with the port as below:





**Figure 3.5 : Git Repo Locally Hosted**

