

INTERNALS QUESTIONS

1. Create a simple **User Registration Form** for an event with fields (Name, Email, Phone).
Initialize a **Git repository**, commit the project files, and push them to GitHub using appropriate Git commands.
2. Create a new **branch** named `update-form`, modify the registration form by adding a new field (Department), merge the branch into `main`, and push the changes to GitHub.
Show the use of: `git branch`, `git checkout`, `git merge`, `git push`.
3. Your college department is developing a small **student portal website** hosted on GitHub. As a DevOps engineer, you must set up **Continuous Integration** using Jenkins

Task Requirements

- Configure a **Jenkins Freestyle Job** that automatically pulls the website code from GitHub.
 - Enable Jenkins to **build the project** every time a change is pushed.
 - Make a small update to the HTML page, push it to GitHub, and show that Jenkins automatically triggers a new build.
4. Create a simple web application (HTML or Python Flask).
Write a **Dockerfile**, build a Docker image, and run the container so the application is accessible on a browser using port mapping.
 5. Using Docker commands, perform the following operations:
 - List images and containers
 - Stop a running container
 - Remove a container and imageDemonstrate commands like: `docker ps`, `docker stop`, `docker rm`, `docker rmi`.
 6. Deploy the previously created Docker image on Kubernetes using a **Deployment YAML file**.
Verify that the pod is running using `kubectl` commands.
 7. Expose the application using a **NodePort Service**, access it using the node's IP and port, and scale the deployment to **3 replicas** using `kubectl scale`.
 8. Set up a simple ML project environment by creating a `requirements.txt` file, installing packages, and verifying environment setup.
Document the steps in a Jupyter Notebook and commit it to the Git repository.

1 — Create a simple User Registration Form; initialize Git, commit, push to GitHub

Files to create:

- index.html
- README.md (optional)

index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <meta name="viewport" content="width=device-width,initial-scale=1"/>
  <title>Event Registration</title>
</head>
<body>
  <h1>Event Registration</h1>
  <form id="regForm" action="#" method="post">
    <label>Name: <input type="text" name="name" required></label><br>
    <label>Email: <input type="email" name="email" required></label><br>
    <label>Phone: <input type="tel" name="phone" required></label><br>
    <button type="submit">Register</button>
  </form>
</body>
</html>
```

Terminal commands (local machine):

```
# create project folder
mkdir student-portal && cd student-portal

# add file
cat > index.html <<'HTML'
... (paste content above) ...
HTML

# initialize Git
git init

# set user.name/user.email if not set (optional)
git config user.name "Your Name"
git config user.email "you@example.com"

# stage & commit
git add index.html
git commit -m "Initial commit: add registration form"
```

```
# create remote repo on GitHub (manually via web UI) and then:
git remote add origin git@github.com:YOUR_USERNAME/student-portal.git
```

```
# push to GitHub
git branch -M main
git push -u origin main
```

Expected verification commands & outputs (examples):

```
# check git log
git log --oneline
# example output:
# ab3c9f2 Initial commit: add registration form

# check remote
git remote -v
# origin  git@github.com:YOUR_USERNAME/student-portal.git (fetch)
# origin  git@github.com:YOUR_USERNAME/student-portal.git (push)
```

2 — Create new branch `update-form`, add `Department` field, merge into `main`, push (show git branch/checkout/merge/push)

Commands & changes:

```
# create and switch to branch
git checkout -b update-form

# modify index.html to add Department field (edit file)
# Insert after Phone line:
# <label>Department: <input type="text" name="department"
# required></label><br>

# stage & commit
git add index.html
git commit -m "Add Department field to registration form"

# switch back to main
git checkout main

# merge branch into main
git merge update-form

# push updated main to remote
git push origin main

# optionally delete branch locally and remotely
git branch -d update-form
git push origin --delete update-form
```

Expected git outputs (examples):

```
$ git checkout -b update-form
Switched to a new branch 'update-form'

$ git commit -m "Add Department field to registration form"
[update-form 6d7e4c1] Add Department field to registration form
1 file changed, 3 insertions(+)
```

```
$ git checkout main
Switched to branch 'main'

$ git merge update-form
Updating ab3c9f2..6d7e4c1
Fast-forward
 index.html | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

$ git push origin main
To github.com:YOUR_USERNAME/student-portal.git
 ab3c9f2..6d7e4c1  main -> main

$ git branch -d update-form
Deleted branch update-form (was 6d7e4c1).
Now index.html includes Department. Verify:

git show HEAD:index.html | sed -n '1,120p'
You should see the new Department <input> line.
```

3 — Jenkins Freestyle Job: auto pull from GitHub, trigger on push, demonstrate build after pushing an HTML change

Overview steps (you will run these in your Jenkins server):

A. **Install Jenkins** (assumes Jenkins already available). Install plugins:

- Git plugin
- GitHub plugin (GitHub Integration / GitHub Hook)
- GitHub Branch Source (optional)
- (For local testing) the "Build periodically" or GitHub webhook integration

B. **Create Jenkins Job:**

1. Jenkins UI → New Item → Freestyle project → name `student-portal-site` → OK.
2. Under **Source Code Management** choose **Git**:
 - Repository URL:

`https://github.com/YOUR_USERNAME/student-portal.git` (or

`git@github.com:...` with credentials).
 - Add credentials (username/password or SSH key) if required.
3. Under **Build Triggers**:
 - Check **GitHub hook trigger for GITScm polling** (or "Build when a change is pushed to GitHub").
4. Under **Build**:
 - Add a build step **Execute shell** with something simple like:


```
echo "Building student-portal"
```

- o `ls -la`
- o `echo "Contents of index.html:"`
- o `sed -n '1,80p' index.html`

5. Save.

C. Configure GitHub webhook:

- Go to GitHub repo → Settings → Webhooks → Add webhook.
 - o Payload URL: `http://<JENKINS_HOST>/github-webhook/` (Jenkins base URL + `/github-webhook/`)
 - o Content type: `application/json`
 - o Which events: **Just the push event**
- Save webhook.

D. Make a small update and push:

```
# make a small change on a new branch or directly on main
git checkout -b tiny-update
# change index.html (e.g., add a comment or update heading)
sed -i "s/Event Registration/Event Registration - Updated/" index.html
git add index.html
git commit -m "Tiny update: change heading for CI test"
git push -u origin tiny-update
```

```
# merge to main (or open PR and merge)
git checkout main
git merge tiny-update
git push origin main
```

E. Expected Jenkins behavior:

- The GitHub webhook sends a push event to Jenkins.
- Jenkins triggers the `student-portal-site` job.
- In Jenkins job console output you will see:

```
Building student-portal
total 8
-rw-r--r-- 1 jenkins jenkins 234 Nov 26 10:00 index.html
Contents of index.html:
<!doctype html>
...
<h1>Event Registration - Updated</h1>
...
Finished: SUCCESS
```

Verification commands on GitHub:

- View webhook deliveries in the repo (Settings → Webhooks) — you should see recent delivery with status 200.
- In Jenkins: check the build history and console output for the triggered build.

4 — Create a simple web application (Flask) + Dockerfile, build image, run container with port mapping

Files:

app.py

```
from flask import Flask, render_template_string
app = Flask(__name__)
```

```
HTML = """
<!doctype html>
<html><body>
  <h1>Student Portal</h1>
  <p>Registration form:</p>
  <form>
    Name: <input name="name"><br>
    Email: <input name="email"><br>
    Phone: <input name="phone"><br>
    Department: <input name="department"><br>
  </form>
</body></html>
"""
```

```
@app.route("/")
def home():
    return render_template_string(HTML)
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
requirements.txt
```

Flask==2.3.2
Dockerfile

```
FROM python:3.11-slim
```

```
WORKDIR /app
```

```
COPY requirements.txt .
```

```
RUN pip install --no-cache-dir -r requirements.txt
```

```
COPY app.py .
```

```
EXPOSE 5000
```

```
CMD ["python", "app.py"]
```

Build and run (commands):

```
# build image
docker build -t student-portal:v1 .
```

```
# run container mapping host port 5000 -> container 5000
docker run -d --name student-portal -p 5000:5000 student-portal:v1
```

Expected outputs:

```
$ docker build -t student-portal:v1 .
[...]  
Successfully tagged student-portal:v1
```

```
$ docker run -d --name student-portal -p 5000:5000 student-portal:v1
```

```
d9f1c3a2b4e7f6a5c3b2...
```

Verify in browser or curl:

```
curl -s http://localhost:5000 | head -n 10
# Example output begins with:
# <!doctype html>
# <html><body>
#   <h1>Student Portal</h1>
#   <p>Registration form:</p>
```

5 — Docker commands: list images/containers, stop container, remove container/image

Commands and expected outputs:

List running containers:

```
docker ps
# Example:
# CONTAINER ID   IMAGE                COMMAND                  CREATED        STATUS
PORTS           NAMES
# d9f1c3a2b4e7   student-portal:v1   "python app.py"        2 minutes ago  Up 2 mins
0.0.0.0:5000->5000/tcp   student-portal
```

List all containers:

```
docker ps -a
```

List images:

```
docker images
# REPOSITORY      TAG    IMAGE ID    CREATED    SIZE
# student-portal  v1     alb2c3d4e5f6 2 hours ago 100MB
```

Stop a running container:

```
docker stop student-portal
# Output: student-portal
```

Remove container:

```
docker rm student-portal
# Output: student-portal
```

Remove image:

```
docker rmi student-portal:v1
# Example output:
# Untagged: student-portal:v1
# Deleted: sha256:alb2c3...
```

6 — Deploy Docker image on Kubernetes using a Deployment YAML; verify pod running

Create deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```

name: student-portal-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: student-portal
  template:
    metadata:
      labels:
        app: student-portal
    spec:
      containers:
        - name: student-portal
          image: YOUR_DOCKER_REGISTRY/student-portal:v1
          ports:
            - containerPort: 5000

```

Notes:

- If image is local to minikube or kind, either load into cluster (minikube image load student-portal:v1) or push to Docker Hub (docker tag ... && docker push ...).

Apply deployment:

```
kubectl apply -f deployment.yaml
```

Verify pod:

```
kubectl get deployments
```

```

# NAME                READY    UP-TO-DATE    AVAILABLE    AGE
# student-portal-deploy 1/1      1             1            10s

```

```
kubectl get pods -l app=student-portal
```

```

# NAME                                     READY    STATUS    RESTARTS    AGE
# student-portal-deploy-66b8c6f5f9-abcde  1/1      Running   0            20s

```

```
kubectl logs <pod-name>
```

```
# Should show Flask started message or your app prints.
```

Example output:

```
$ kubectl get pods
```

```

NAME                                     READY    STATUS    RESTARTS
AGE
student-portal-deploy-66b8c6f5f9-abcde  1/1      Running   0
30s

```

7 — Expose application with NodePort and scale to 3 replicas

Create service-nodeport.yaml:

```

apiVersion: v1
kind: Service
metadata:
  name: student-portal-nodeport
spec:
  type: NodePort
  selector:

```



```

    app: student-portal
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
      nodePort: 30080    # optional; if omitted, Kubernetes picks a port in
                          30000-32767

```

Apply service:

```

kubectl apply -f service-nodeport.yaml
kubectl get svc student-portal-nodeport
# NAME                                TYPE             CLUSTER-IP       EXTERNAL-IP       PORT(S)
AGE
# student-portal-nodeport             NodePort         10.96.x.y         <none>
5000:30080/TCP    10s

```

Find node IP (for a single-node cluster, use the node's IP; for minikube you can use `minikube ip`):

```

# On Linux / cloud K8s, get nodes:
kubectl get nodes -o wide
# Use the EXTERNAL-IP or INTERNAL-IP column value

```

Access service:

```

# Example if node ip is 192.168.1.100 and nodePort 30080
curl http://192.168.1.100:30080 | head -n 10
# you should see the HTML response of the Flask app

```

Scale deployment to 3 replicas:

```

kubectl scale deployment student-portal-deploy --replicas=3
kubectl get pods -l app=student-portal
# You should see 3 pods in Running state:
# student-portal-deploy-...    1/1    Running
# student-portal-deploy-...    1/1    Running
# student-portal-deploy-...    1/1    Running

```

Expected outputs:

```

$ kubectl get pods
NAME                                READY   STATUS    RESTARTS
AGE
student-portal-deploy-66b8c6f5f9-abcde    1/1     Running   0
1m
student-portal-deploy-66b8c6f5f9-bghij    1/1     Running   0
10s
student-portal-deploy-66b8c6f5f9-klmno    1/1     Running   0
10s

$ curl http://192.168.1.100:30080
<!doctype html>...

```

8 — Set up simple ML project environment, create `requirements.txt`, install packages, document steps in a Jupyter Notebook, commit to Git

Files:

- `requirements-ml.txt` (or same `requirements.txt`)

```
numpy==1.26.2
pandas==2.2.2
scikit-learn==1.3.2
jupyterlab==4.1.0
```

Set up virtual environment & install:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements-ml.txt
```

Verify installed packages:

```
pip freeze | grep -E "numpy|pandas|scikit-learn|jupyter"
# Example output:
# numpy==1.26.2
# pandas==2.2.2
# scikit-learn==1.3.2
# jupyterlab==4.1.0
```

Create `ml_setup.ipynb` (Jupyter Notebook) — minimal notebook content (you can create via Jupyter Notebook UI or programmatically). Example cells to include:

Cell 1 — Markdown:

```
# ML Environment Setup
This notebook verifies that required Python packages are installed and runs a
tiny sample.
```

Cell 2 — Code:

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

print("numpy:", np.__version__)
print("pandas:", pd.__version__)

# tiny sample
X = np.array([[1],[2],[3],[4]])
y = np.array([2,4,6,8])
model = LinearRegression().fit(X, y)
print("coef:", model.coef_, "intercept:", model.intercept_)
```

Run the notebook (locally or on Jupyter). Expected output:

```
numpy: 1.26.2
pandas: 2.2.2
coef: [2.] intercept: 0.0
```

Commit Jupyter notebook and requirements to Git:

```
git add requirements-ml.txt ml_setup.ipynb
git commit -m "Add ML environment requirements and verification notebook"
git push origin main
```