Experiment No: 5

Title: Demonstrate Continuous Integration and Development using Jenkins

Aim: To demonstrate Continuous Integration (CI) and Continuous Development (CD) using Jenkins.

Description:

Continuous Integration (CI) and Continuous Development (CD) are important practices in software development that automate the process of building, testing, and deploying code. Jenkins, an open-source automation server, is widely used to achieve CI/CD.

The following steps demonstrate how to use Jenkins for CI/CD:

1. **Create a simple Java application:**
   - Develop a small Java program (e.g., printing "Hello World" or performing simple calculations).
2. **Commit the code to a Git repository:**
   - Create a Git repository and commit your Java project.
   - Ensure the Jenkins server can access the repository.
3. **Create a Jenkins Job:**
   - Log in to Jenkins and create a new job (Freestyle or Pipeline).
   - Configure the job to pull code from the Git repository.
   - Set build triggers (for example, build automatically after each commit).
4. **Build the Application:**
   - Run the Jenkins job to compile the Java code, run tests, and generate an executable JAR file.
5. **Monitor the Build:**
   - Observe the progress and logs in Jenkins.
   - View build results and reports (if applicable).
6. **Deploy the Application:**

- If the build succeeds, configure Jenkins to deploy the JAR file to a test or production server automatically.

This simple workflow demonstrates how Jenkins can automate the continuous integration and deployment of software. In real-world environments, CI/CD pipelines can involve multiple stages, such as code analysis, testing, staging, and production deployment.

Experiment No: 6

Title: Explore Docker Commands for Content Management

Aim: To explore and use Docker commands for managing containers and images.

Description: Docker is a containerization platform used to package, deploy, and run applications in isolated environments called containers. It provides various commands to manage containers and images efficiently.

Commonly used Docker commands:

Command	**Description**	**Example**
docker run	Runs a command in a new container	docker run --name mycontainer -it ubuntu:16.04 /bin/bash
docker start	Starts one or more stopped containers	docker start mycontainer
docker stop	Stops one or more running containers	docker stop mycontainer
docker rm	Removes one or more containers	docker rm mycontainer
docker ps	Lists running containers	docker ps
docker images	Lists all local images	docker images
docker pull	Pulls an image from a registry	docker pull ubuntu:16.04
docker push	Pushes an image to a registry	docker push myimage

These are basic Docker commands for content management. Advanced commands are available for managing networks, volumes, and configurations.

Experiment No: 7

Title: Develop a Simple Containerized Application using Docker

Aim:

To develop and run a simple containerized application using Docker.

Description:

This experiment demonstrates how to create and run a simple containerized application using Docker.

Steps:

1. **Choose an application:**
   Create a simple Python script named hello.py that prints “Hello World”.

   ```
   print("Hello World from Docker container!")
   ```

2. **Write a Docker file:**
   Create a file named Docker file in the same directory as hello.py:

   ```
   # Use the official Python image as base

   FROM python:3.9

   # Copy script into the container

   COPY hello.py /app/

   # Set working directory

   WORKDIR /app/

   # Command to run the script

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

3. **Build the Docker image**

   ```
   docker build -t myimage .
   ```

4. **Run the Docker container:**

   ```
   docker run --name mycontainer myimage
   ```

5. **Verify the output:**

   ```
   docker logs mycontainer
   ```

Expected Output:

```
Hello World from Docker container!
```

Experiment No: 8

Title: Integrate Kubernetes and Docker

Aim:

To integrate Kubernetes and Docker for container orchestration.

Description:

Docker is used for containerizing applications, while Kubernetes is used for orchestrating and managing these containers across a cluster. Integration of Docker and Kubernetes involves deploying Docker images into a Kubernetes cluster.

Steps:

- **Build a Docker image:**
  Use Docker to build a container image of your application.

```
docker build -t myapp .
```

- **Push the image to a registry:**
  Push the Docker image to a registry like Docker Hub.

```
docker push myapp
```

- **Create a Deployment:**
  Write a deployment YAML file (deployment.yaml):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myimage
        ports:
        - containerPort: 80
```

- Create **a Service:**
  Write a service YAML file (service.yaml):

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
  - name: http
    port: 80
    targetPort: 80
  type: ClusterIP
```

Deploy to Kubernetes:

bash

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

```
kubectl apply -f service.yaml
```

Monitor and Manage:

bash

Copy code

```
kubectl get pods
```

```
kubectl get services
```

Kubernetes will schedule and manage the containers created from the Docker image, providing scalability and high availability.

Experiment No: 9

Title: Automate the Process of Running Containerized Application using Kubernetes

Aim:

To automate the process of running the containerized application (developed in Experiment 7) using Kubernetes.

Description:

This experiment demonstrates how to automate deployment and management of a Dockerized application using Kubernetes.

Steps:

1. **Create a Kubernetes Cluster:**
   Use Minikube or a cloud provider (Google Kubernetes Engine, AWS EKS, etc.).

2. **Push the Docker Image to Registry:**

```
docker push myimage
```

3. **Create Deployment File (deployment.yaml):**

```
apiVersion: apps/v1
```

```yaml
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myimage
        ports:
        - containerPort: 80
```

Create Service File (service.yaml):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
  - name: http
    port: 80
    targetPort: 80
  type: ClusterIP
```

Apply the Deployment and Service:

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

Verify Deployment:

```
kubectl get pods
kubectl get services
```

Kubernetes automates the running, scaling, and management of the containerized application.