

1. Setting Up Environment Install Maven using the commands below: `sudo apt update` `sudo apt install maven` You can find the details of the Jenkins credentials and MySQL database local passwords in the Readme.txt file on the Desktop. Add Jenkins to the Docker group and restart Jenkins. Note: Use the docker compose command instead of the legacy docker-compose, as it is integrated into Docker CLI (v20.10+), providing better performance, consistency, and active maintenance, unlike the legacy command.

To set up the environment for deploying a multi-tier Java application to Kubernetes, follow the steps below. This includes installing Maven, configuring Jenkins, and ensuring Docker is set up correctly.

1. Install Maven

Maven is required to build the Java application. Install it using the following commands:

```
bash
sudo apt update
sudo apt install maven
```

Verify the installation:

```
bash
mvn -version
```

2. Configure Jenkins

Jenkins will be used for CI/CD pipelines. Ensure Jenkins is installed and configured properly.

Add Jenkins to the Docker Group

To allow Jenkins to interact with Docker, add the Jenkins user to the Docker group:

```
bash
sudo usermod -aG docker jenkins
```

Restart Jenkins to apply the changes:

```
bash
sudo systemctl restart jenkins
```

Verify Jenkins Docker Access

Log in to the Jenkins server and verify that Jenkins can run Docker commands by executing a simple test script in a Jenkins pipeline:

```
groovy
pipeline {
  agent any
  stages {
    stage("Test Docker") {
```

```
    steps {  
      sh 'docker --version'  
    }  
  }  
}
```

3. Retrieve Jenkins Credentials and MySQL Passwords

The `Readme.txt` file on the Desktop contains the Jenkins credentials and MySQL database passwords. Use these credentials to configure Jenkins and MySQL.

Example Readme.txt Content:

Copy

Jenkins Credentials:

- Username: admin
- Password: admin123

MySQL Database:

- Username: root
- Password: mysqlpassword

4. Use Docker Compose (Integrated into Docker CLI)

The `docker compose` command is now integrated into the Docker CLI (v20.10+). It replaces the legacy `docker-compose` command and provides better performance and consistency.

Verify Docker Compose Version

Ensure Docker Compose is installed and check its version:

```
bash  
docker compose version
```

Example Docker Compose File

Create a `docker-compose.yml` file to define your multi-tier application (e.g., Java backend, MySQL database):

```
yaml  
version: '3.8'  
services:  
  mysql:  
    image: mysql:5.7  
    environment:  
      MYSQL_ROOT_PASSWORD: mysqlpassword  
      MYSQL_DATABASE: mydb  
    volumes:  
      - mysql-data:/var/lib/mysql  
    networks:  
      - app-network  
  
  java-backend:  
    image: your-registry/your-username/java-backend:latest
```

```
environment:
  SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/mydb
  SPRING_DATASOURCE_USERNAME: root
  SPRING_DATASOURCE_PASSWORD: mysqlpassword
ports:
  - "8080:8080"
depends_on:
  - mysql
networks:
  - app-network

volumes:
  mysql-data:

networks:
  app-network:
```

Run Docker Compose

Start the application using the docker compose command:

```
bash
docker compose up -d
```

5. Verify the Setup

- **Check Running Containers:**

```
bash
docker ps
```

- **Access the Java Application:**
Open a browser and navigate to <http://localhost:8080> to verify the Java backend is running.
- **Check MySQL Connection:**
Connect to the MySQL database using the credentials from the Readme.txt file:

```
bash
mysql -h 127.0.0.1 -u root -p
```

6. Prepare for Kubernetes Deployment

Once the application is running locally with Docker Compose, you can proceed to deploy it to Kubernetes. Follow these steps:

1. **Push Docker Images to a Registry:**
Push the Java backend and MySQL images to a container registry (e.g., Docker Hub, GCR, ECR).
2. **Create Kubernetes Manifests:**
Write Kubernetes manifests (Deployment, Service, ConfigMap, etc.) for each component.
3. **Deploy to Kubernetes:**
Use `kubectl apply` to deploy the application to your Kubernetes cluster.

7. Notes

- **Docker Compose vs. Kubernetes:**
Docker Compose is great for local development and testing, while Kubernetes is ideal for production deployments.
- **Security:**
Ensure sensitive data (e.g., MySQL passwords) is stored securely using Kubernetes Secrets.
- **Jenkins Pipelines:**
Automate the build, test, and deployment process using Jenkins pipelines.

By following these steps, you will have a fully functional environment for developing, testing, and deploying a multi-tier Java application.

[New chat](#)

1. Setting Up Environment Install Maven using the commands below: `sudo apt update` `sudo apt install maven` You can find the details of the Jenkins credentials and MySQL database local passwords in the Readme.txt file on the Desktop. Add Jenkins to the Docker group and restart Jenkins. Note: Use the `docker compose` command instead of the legacy `docker-compose`, as it is integrated into Docker CLI (v20.10+), providing better performance, consistency, and active maintenance, unlike the legacy command.

Here's a step-by-step guide to set up your environment based on the instructions provided:

1. Update and Install Maven

Run the following commands to update your package list and install Maven:

```
bash
sudo apt update
sudo apt install maven -y
```

2. Retrieve Jenkins Credentials and MySQL Passwords

- Navigate to the Readme.txt file on your Desktop to find the Jenkins credentials and MySQL database local passwords.
- Use a text editor or `cat` command to view the file:

```
bash
cat ~/Desktop/Readme.txt
```

3. Add Jenkins to the Docker Group

To allow Jenkins to interact with Docker, add the Jenkins user to the Docker group:

```
bash
```

```
sudo usermod -aG docker jenkins
```

After adding Jenkins to the Docker group, restart the Jenkins service:

```
bash
sudo systemctl restart jenkins
```

4. Use docker compose Instead of docker-compose

- Ensure you are using Docker CLI version 20.10 or higher.
- Use the `docker compose` command (with a space) instead of the legacy `docker-compose` (with a hyphen) for better performance and consistency.

Example:

```
bash
docker compose up -d
```

5. Verify Docker Compose Version

To confirm your Docker CLI version and ensure `docker compose` is available, run:

```
bash
docker --version
docker compose version
```

6. Additional Notes

- If Docker is not installed, install it using the official Docker installation guide for your OS.
- Ensure Docker is running before using `docker compose` commands:

```
bash
sudo systemctl status docker
```

Let me know if you need further assistance!

[New chat](#)

2. Building the Docker Image and docker-compose.yaml

1. Navigate to the **Project** folder on the Desktop. Open the **Dockerfile** using Visual Studio Code in the VM lab.
2. Build the project using ***maven:3.9.9-eclipse-temurin-17***.

3. Run the project using **openjdk:17-slim**.
4. Expose the application port on port 8081.
5. Build the image with the name “**todo-application-image**” and tag **latest**.
6. In the **docker-compose.yaml**, the service names of the components should be **todo-application** and **mysql-db**.
7. The local host port and container port should be **8082:8081** for the **todo-application** service.
8. Use “**root**” as the username and “**Root@123**” as password for mysql server.
9. Create a custom bridge network named **todo-network**.
10. Use database name as “**tododb**”.
11. The local host port and container port should be **3307:3306** for the **mysql-db** service.
12. Use volumes to persist data. The volume name should be “**mysql-data**”.
13. Create a repository with the name “**todo-application**” in Docker Hub using this [link](#).

Below are the detailed steps to build the Docker image, configure the docker-compose.yaml file, and push the image to Docker Hub as per your requirements.

1. Navigate to the Project Folder

Open the terminal and navigate to the project folder on the Desktop:

```
bash
cd ~/Desktop/Project
```

2. Open the Dockerfile in Visual Studio Code

Open the Dockerfile using Visual Studio Code:

```
bash
code Dockerfile
```

3. Build the Project Using Maven

Ensure the project is built using the maven:3.9.9-eclipse-temurin-17 image. Add the following to your Dockerfile:

```
Dockerfile
# Stage 1: Build the project using Maven
FROM maven:3.9.9-eclipse-temurin-17 AS build
WORKDIR /app
COPY . .
RUN mvn clean package -DskipTests

# Stage 2: Run the project using openjdk:17-slim
FROM openjdk:17-slim
```

```
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8081
ENTRYPOINT ["java", "-jar", "app.jar"]
```

4. Build the Docker Image

Build the Docker image with the name `todo-application-image` and tag it as `latest`:

```
bash
docker build -t todo-application-image:latest .
```

5. Create the docker-compose.yaml File

Create a `docker-compose.yaml` file in the project directory with the following configuration:

```
yaml
version: '3.8'
services:
  todo-application:
    image: todo-application-image:latest
    container_name: todo-application
    ports:
      - "8082:8081"
    networks:
      - todo-network
    depends_on:
      - mysql-db
  mysql-db:
    image: mysql:5.7
    container_name: mysql-db
    environment:
      MYSQL_ROOT_PASSWORD: Root@123
      MYSQL_DATABASE: tododb
    ports:
      - "3307:3306"
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - todo-network
volumes:
  mysql-data:
networks:
  todo-network:
    driver: bridge
```

6. Run the Application Using Docker Compose

Start the application using the `docker compose` command:

```
bash
docker compose up -d
```

7. Verify the Application

- **Check Running Containers:**

```
bash
```

```
docker ps
```

- **Access the Todo Application:**

Open a browser and navigate to <http://localhost:8082> to verify the application is running.

- **Check MySQL Connection:**

Connect to the MySQL database using the credentials:

```
bash
```

```
mysql -h 127.0.0.1 -P 3307 -u root -p
```

8. Push the Docker Image to Docker Hub

1. **Log in to Docker Hub:**

```
bash
```

```
docker login
```

Enter your Docker Hub username and password when prompted.

2. **Tag the Docker Image:**

Tag the todo-application-image with your Docker Hub repository name:

```
bash
```

```
docker tag todo-application-image:latest <your-dockerhub-username>/todo-application:latest
```

3. **Push the Image to Docker Hub:**

Push the image to the todo-application repository:

```
bash
```

```
docker push <your-dockerhub-username>/todo-application:latest
```

9. Verify the Image on Docker Hub

1. Go to [Docker Hub](https://hub.docker.com/).
2. Log in to your account.
3. Navigate to the todo-application repository.
4. Verify that the image has been successfully pushed.

10. Clean Up

If you want to stop and remove the containers, run:

```
bash
```


Summary

- **Dockerfile:** Used to build the Java application using Maven and run it with OpenJDK 17.
- **docker-compose.yaml:** Defines the todo-application and mysql-db services, exposes ports, and configures the network and volumes.
- **Docker Hub:** The todo-application-image is pushed to Docker Hub for future use.

By following these steps, you will have a fully functional multi-tier Java application running locally with Docker Compose and its image stored in Docker Hub for deployment to Kubernetes or other environments.

[New chat](#)

3. Setting Up GitHub Repository and Docker Hub

1. Create a public repository in your personal GitHub account using the provided link.
2. Ensure the repository is publicly accessible. If it is private, generate a Personal Access Token (PAT) to access it. Update the changes in the **Jenkinsfile** with the GitHub credentials. The project should be in the **master** branch.
3. Once all updates with your Docker Hub and GitHub details are complete, clone the application folder or project folder to the GitHub repository created earlier using the Linux terminal.

Below are the steps to set up a GitHub repository, configure Docker Hub, and clone the project folder to the GitHub repository. Additionally, I'll guide you on updating the Jenkinsfile with GitHub credentials if the repository is private.

1. Create a Public GitHub Repository

1. Go to [GitHub](#) and log in to your account.
 2. Click the + icon in the top-right corner and select **New repository**.
 3. Fill in the repository details:
 - **Repository name:** todo-application
 - **Visibility:** Select **Public** (if private, generate a Personal Access Token as described below).
 - Click **Create repository**.
-

2. Generate a Personal Access Token (PAT) for Private Repositories

If your repository is private, generate a PAT to allow Jenkins to access it:

1. Go to your GitHub account settings.
2. Navigate to **Developer settings** > **Personal access tokens** > **Tokens (classic)**.
3. Click **Generate new token**.
4. Select the required scopes (e.g., repo, admin:repo_hook).
5. Click **Generate token**.

6. Copy the token and save it securely (you won't be able to see it again).

3. Update Jenkinsfile with GitHub Credentials

If your repository is private, update the Jenkinsfile to include the GitHub credentials (PAT).

Example Jenkinsfile:

```
groovy
pipeline {
  agent any
  environment {
    GITHUB_CREDENTIALS = credentials('github-pat') // Add GitHub PAT as a Jenkins credential
  }
  stages {
    stage('Checkout') {
      steps {
        git(
          url: 'https://github.com/<your-github-username>/todo-application.git',
          branch: 'master',
          credentialsId: 'github-pat' // Use the credential ID for GitHub PAT
        )
      }
    }
    stage('Build and Push Docker Image') {
      steps {
        script {
          sh 'docker build -t todo-application-image:latest .'
          sh 'docker tag todo-application-image:latest <your-dockerhub-username>/todo-application:latest'
          withCredentials([usernamePassword(credentialsId: 'dockerhub-creds', usernameVariable: 'DOCKER_USERNAME', passwordVariable: 'DOCKER_PASSWORD')]) {
            sh 'docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD'
            sh 'docker push <your-dockerhub-username>/todo-application:latest'
          }
        }
      }
    }
  }
}
```

Add GitHub PAT to Jenkins:

1. Go to Jenkins > **Manage Jenkins** > **Manage Credentials**.
2. Add a new credential:
 - o **Kind:** Secret text
 - o **Secret:** Paste your GitHub PAT.
 - o **ID:** github-pat (use this ID in the Jenkinsfile).

4. Clone the Project Folder to GitHub Repository

1. Open the terminal and navigate to the project folder:

```
bash
```

```
cd ~/Desktop/Project
```

2. Initialize a Git repository:

```
bash
```

```
git init
```

3. Add all files to the repository:

```
bash
```

```
git add .
```

4. Commit the changes:

```
bash
```

```
git commit -m "Initial commit"
```

5. Add the remote GitHub repository:

```
bash
```

```
git remote add origin https://github.com/<your-github-username>/todo-application.git
```

6. Push the changes to the master branch:

```
bash
```

```
git branch -M master
```

```
git push -u origin master
```

5. Verify the GitHub Repository

1. Go to your GitHub repository (<https://github.com/<your-github-username>/todo-application>).
2. Verify that all files have been pushed successfully.

6. Configure Docker Hub Credentials in Jenkins

If you haven't already, add your Docker Hub credentials to Jenkins:

1. Go to Jenkins > **Manage Jenkins** > **Manage Credentials**.
2. Add a new credential:
 - **Kind:** Username and password
 - **Username:** Your Docker Hub username.
 - **Password:** Your Docker Hub password or access token.
 - **ID:** dockerhub-creds (use this ID in the Jenkinsfile).

7. Run the Jenkins Pipeline

1. Create a new pipeline job in Jenkins.
2. Configure the pipeline to use the Jenkinsfile from the GitHub repository.
3. Run the pipeline to build and push the Docker image to Docker Hub.

Summary

- **GitHub Repository:** Created a public repository and cloned the project folder.
- **Jenkinsfile:** Updated with GitHub credentials (if private) and Docker Hub credentials.
- **Docker Hub:** Configured Jenkins to push the Docker image to Docker Hub.

By following these steps, you will have a fully automated CI/CD pipeline that builds the Docker image, pushes it to Docker Hub, and integrates with GitHub for version control.

[New chat](#)

4. Configuring Jenkins Pipeline

1. Go to Manage Jenkins > Credentials > Global > Add Credentials.
2. Add your Docker Hub credentials and save them with the ID “**docker-hub-credentials.**”
3. If the GitHub repository created is private, you will need to add the credentials for GitHub.
4. Create a new Jenkins pipeline named “**todo-application-pipeline**”.
5. Clone the repository from GitHub and build the project with Maven by skipping tests.
6. Build and push the Docker image to Docker Hub using Docker Hub credentials.
7. Deploy the application using Docker Compose and verify the services.
8. Clean the workspace using “**rm -rf ***”.
9. Once the **Jenkinsfile** is configured, clone the repository into the previously created GitHub repository.
10. After a successful build, the Maven app will be visible on **port 8082** in Chrome within the VM lab.

Below are the detailed steps to configure the Jenkins pipeline for your todo-application. This includes setting up credentials, creating the pipeline, and configuring the Jenkinsfile to build the project, push the Docker image, and deploy the application using Docker Compose.

1. Add Docker Hub Credentials to Jenkins

1. Go to **Manage Jenkins > Manage Credentials.**
 2. Under **Global credentials**, click **Add Credentials.**
 3. Fill in the details:
 - **Kind:** Username and password
 - **Username:** Your Docker Hub username.
 - **Password:** Your Docker Hub password or access token.
 - **ID:** docker-hub-credentials (use this ID in the Jenkinsfile).
 - **Description:** Docker Hub credentials.
 4. Click **OK** to save.
-

2. Add GitHub Credentials (if the Repository is Private)

1. Go to **Manage Jenkins > Manage Credentials**.
 2. Under **Global credentials**, click **Add Credentials**.
 3. Fill in the details:
 - **Kind**: Secret text
 - **Secret**: Your GitHub Personal Access Token (PAT).
 - **ID**: github-pat (use this ID in the Jenkinsfile).
 - **Description**: GitHub PAT for private repository access.
 4. Click **OK** to save.
-

3. Create a New Jenkins Pipeline

1. Go to the Jenkins dashboard and click **New Item**.
 2. Enter the name `todo-application-pipeline` and select **Pipeline**.
 3. Click **OK**.
-

4. Configure the Pipeline

1. In the pipeline configuration page:
 - Under **Pipeline**, select **Pipeline script from SCM**.
 - **SCM**: Select **Git**.
 - **Repository URL**: Enter your GitHub repository URL (e.g., `https://github.com/<your-github-username>/todo-application.git`).
 - **Credentials**: Select the GitHub credentials (`github-pat`) if the repository is private.
 - **Branch Specifier**: Enter `*/master`.
 - **Script Path**: Enter `Jenkinsfile` (this is the path to your Jenkinsfile in the repository).
 2. Click **Save**.
-

5. Configure the Jenkinsfile

Create a Jenkinsfile in your project directory with the following content:

```
groovy
Copy
pipeline {
    agent any
    environment {
        DOCKER_IMAGE = "todo-application-image"
        DOCKER_TAG = "latest"
        DOCKER_HUB_CREDENTIALS = "docker-hub-credentials"
    }
    stages {
        stage('Checkout') {
            steps {
                git(
                    url: 'https://github.com/<your-github-username>/todo-application.git',
                    branch: 'master',
                    credentialsId: 'github-pat' // Only if the repository is private
                )
            }
        }
    }
}
```

```

    }
    stage('Build with Maven') {
        steps {
            sh 'mvn clean package -DskipTests'
        }
    }
    stage('Build and Push Docker Image') {
        steps {
            script {
                sh "docker build -t ${DOCKER_IMAGE}:${DOCKER_TAG} ."
                withCredentials([usernamePassword(credentialsId: DOCKER_HUB_CREDENTIALS, usernameVariable: 'DOCKER_USERNAME', passwordVariable: 'DOCKER_PASSWORD')]) {
                    sh "docker tag ${DOCKER_IMAGE}:${DOCKER_TAG} ${DOCKER_USERNAME}/${DOCKER_IMAGE}:${DOCKER_TAG}"
                }
                sh "docker login -u ${DOCKER_USERNAME} -p ${DOCKER_PASSWORD}"
                sh "docker push ${DOCKER_USERNAME}/${DOCKER_IMAGE}:${DOCKER_TAG}"
            }
        }
    }
    stage('Deploy with Docker Compose') {
        steps {
            sh 'docker compose up -d'
        }
    }
    stage('Verify Services') {
        steps {
            sh 'docker ps'
            sh 'curl http://localhost:8082' // Verify the application is running
        }
    }
    stage('Clean Workspace') {
        steps {
            sh 'rm -rf *' // Clean the workspace
        }
    }
}

```

6. Push the Jenkinsfile to GitHub

1. Add the Jenkinsfile to your project folder:

```
bash
```

```
Copy
```

```

cd ~/Desktop/Project
git add Jenkinsfile
git commit -m "Added Jenkinsfile for CI/CD pipeline"
git push origin master

```

7. Run the Jenkins Pipeline

1. Go to the Jenkins dashboard and open the todo-application-pipeline.
2. Click **Build Now** to trigger the pipeline.
3. Monitor the pipeline execution in the **Build History** section.

8. Verify the Application

1. Once the pipeline completes successfully, open a browser in the VM lab.
 2. Navigate to <http://localhost:8082> to verify the application is running.
-

9. Clean Up

If you want to stop and remove the Docker containers, run:

```
bash
Copy
docker compose down
```

Summary

- **Jenkins Pipeline:** Configured to clone the repository, build the project with Maven, build and push the Docker image, and deploy the application using Docker Compose.
- **Docker Hub:** The Docker image is pushed to Docker Hub using the credentials stored in Jenkins.
- **GitHub:** The Jenkinsfile is pushed to the GitHub repository for version control.

By following these steps, you will have a fully automated CI/CD pipeline for your `todo-application`. The application will be built, tested, and deployed automatically whenever changes are pushed to the `master` branch.

[New chat](#)

5. Deploy the application using Kubernetes

1. Start Minikube.
2. Navigate to the **Project** folder, where you can find **`todo-application-deployment.yaml`** and **`mysql-deployment.yaml`** files.
3. You can use these files as they are preconfigured as part of the assessment.
4. Create a secret in **kubectl** with the name “**`my-registry-secret`**” to hold the details of Docker Hub credentials.
5. Replace the image details with the details of your Docker Hub repository.
6. Expose Node port **30080**.
7. Apply the files and the application can be available on <http://<minikube-ip>:30080>.

Below are the steps to deploy the `todo-application` to Kubernetes using Minikube. This includes starting Minikube, creating a Kubernetes secret for Docker Hub credentials, updating the deployment files, and exposing the application via a NodePort.

1. Start Minikube

Start Minikube to create a local Kubernetes cluster:

```
bash
Copy
minikube start
```

Verify that Minikube is running:

```
bash
Copy
kubectl get nodes
```

2. Navigate to the Project Folder

Navigate to the folder containing the Kubernetes deployment files (todo-application-deployment.yaml and mysql-deployment.yaml):

```
bash
Copy
cd ~/Desktop/Project
```

3. Create a Kubernetes Secret for Docker Hub Credentials

Create a secret named my-registry-secret to store your Docker Hub credentials:

```
bash
Copy
kubectl create secret docker-registry my-registry-secret \
  --docker-server=https://index.docker.io/v1/ \
  --docker-username=<your-dockerhub-username> \
  --docker-password=<your-dockerhub-password> \
  --docker-email=<your-email>
```

Replace <your-dockerhub-username>, <your-dockerhub-password>, and <your-email> with your Docker Hub credentials.

4. Update the Deployment Files

Update the todo-application-deployment.yaml file to use your Docker Hub image and the secret.

Example todo-application-deployment.yaml:

```
yaml
Copy
apiVersion: apps/v1
kind: Deployment
```



```

metadata:
  name: todo-application
spec:
  replicas: 3
  selector:
    matchLabels:
      app: todo-application
  template:
    metadata:
      labels:
        app: todo-application
    spec:
      containers:
      - name: todo-application
        image: <your-dockerhub-username>/todo-application:latest
        ports:
        - containerPort: 8081
      imagePullSecrets:
      - name: my-registry-secret
---
apiVersion: v1
kind: Service
metadata:
  name: todo-application-service
spec:
  selector:
    app: todo-application
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8081
  type: NodePort
  nodePort: 30080

```

Example mysql-deployment.yaml:

```

yaml
Copy
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql-db
  template:
    metadata:
      labels:
        app: mysql-db
    spec:
      containers:
      - name: mysql-db
        image: mysql:5.7
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: Root@123
        - name: MYSQL_DATABASE
          value: tododb
        ports:
        - containerPort: 3306

```

```
  volumeMounts:
  - name: mysql-data
    mountPath: /var/lib/mysql
volumes:
- name: mysql-data
  persistentVolumeClaim:
    claimName: mysql-pvc
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-db-service
spec:
  selector:
    app: mysql-db
  ports:
  - protocol: TCP
    port: 3306
    targetPort: 3306
```

5. Apply the Deployment Files

Apply the Kubernetes manifests to deploy the application:

```
bash
Copy
kubectl apply -f mysql-deployment.yaml
kubectl apply -f todo-application-deployment.yaml
```

6. Verify the Deployment

Check the status of the deployments, pods, and services:

```
bash
Copy
kubectl get deployments
kubectl get pods
kubectl get services
```

7. Access the Application

1. Get the Minikube IP:

```
bash
```

```
Copy
```

```
minikube ip
```

2. Open a browser and navigate to <http://<minikube-ip>:30080> to access the todo-application.
-

8. Clean Up

If you want to delete the deployments and services, run:

```
bash
```

```
Copy
```

```
kubectl delete -f todo-application-deployment.yaml
```

```
kubectl delete -f mysql-deployment.yaml
```

Summary

- **Minikube:** Used to create a local Kubernetes cluster.
- **Kubernetes Secret:** Created to store Docker Hub credentials.
- **Deployment Files:** Updated with your Docker Hub image and applied to Kubernetes.
- **NodePort:** Exposed the application on port 30080.

By following these steps, you will have successfully deployed the todo-application to Kubernetes using Minikube. The application will be accessible at <http://<minikube-ip>:30080>.

Checking if the user 'jenkins' is part of the 'docker' group. [5 marks]

Checking if the Docker image 'todo-application-image' is created. [5 marks]

Checking if the Docker image 'todo-application-image' is tagged correctly with latest. [5 marks]

Checking if the Docker volume 'mysql-data' is created after executing the Jenkins pipeline. [5 marks]

Checking if the Docker network 'todo-network' is created after executing the Jenkins pipeline. [5 marks]

Checking if the application created as part of the Jenkins pipeline execution is active in the Docker container. [10 marks]

Checking if the MySQL database container 'mysql-db-1' was created and exists as part of the Jenkins pipeline execution. [5 marks]

Checking if the To-Do application container 'todo-application-1' was created and exists as part of the Jenkins pipeline execution. [5 marks]

Checking if the Kubernetes secret with the name 'my-registry-secret' exists. [5 marks]

Checking if the application is running on Minikube at port 30080. [10 marks]

Checking if the application hosted on Minikube at port 30080 serves the expected content. [15 marks]

Checking if the latest build of 'todo-application-pipeline' is successful. [10 marks]

Checking if all required application pods are in a Running state. [10 marks]

Checking if the database 'tododb' exists in the MySQL container. [5 marks]

G

Below is a detailed checklist to verify the tasks mentioned in your assessment. Each task is accompanied by the commands or steps required to validate it.

1. Check if the User 'jenkins' is Part of the 'docker' Group

Run the following command to check if the jenkins user is part of the docker group:

```
bash
groups jenkins
```

Expected Output: jenkins should be listed in the docker group.

2. Check if the Docker Image todo-application-image is Created

List all Docker images to verify if todo-application-image exists:

```
bash
docker images
```

Expected Output: The todo-application-image should be listed.

3. Check if the Docker Image todo-application-image is Tagged Correctly with latest

Verify the tag of the Docker image:

```
bash
docker images | grep todo-application-image
```

Expected Output: The TAG column should show latest.

4. Check if the Docker Volume mysql-data is Created

List all Docker volumes to verify if mysql-data exists:

```
bash
docker volume ls
```

Expected Output: The mysql-data volume should be listed.

5. Check if the Docker Network todo-network is Created

List all Docker networks to verify if todo-network exists:

```
bash
docker network ls
```

Expected Output: The todo-network should be listed.

6. Check if the Application is Active in the Docker Container

Check the status of the running containers:

```
bash
docker ps
```

Expected Output: The todo-application container should be in the Up state.

7. Check if the MySQL Database Container mysql-db-1 Exists

Verify if the MySQL container is running:

```
bash
docker ps | grep mysql-db-1
```

Expected Output: The mysql-db-1 container should be listed.

8. Check if the To-Do Application Container todo-application-1 Exists

Verify if the To-Do application container is running:

```
bash
docker ps | grep todo-application-1
```

Expected Output: The todo-application-1 container should be listed.

9. Check if the Kubernetes Secret my-registry-secret Exists

List all Kubernetes secrets to verify if my-registry-secret exists:

```
bash
kubectl get secrets
```

Expected Output: The my-registry-secret should be listed.

10. Check if the Application is Running on Minikube at Port 30080

1. Get the Minikube IP:

```
bash
minikube ip
```

2. Open a browser and navigate to <http://<minikube-ip>:30080>.

Expected Output: The To-Do application should be accessible.

11. Check if the Application Serves the Expected Content

Verify the content served by the application:

```
bash
curl http://<minikube-ip>:30080
```

Expected Output: The response should include the expected HTML or JSON content.

12. Check if the Latest Build of todo-application-pipeline is Successful

1. Go to the Jenkins dashboard.
2. Open the todo-application-pipeline job.
3. Check the status of the latest build.

Expected Output: The build should be marked as **Successful**.

13. Check if All Required Application Pods are in a Running State

List all pods and their status:

```
bash
kubectl get pods
```

Expected Output: All pods (e.g., todo-application and mysql-db) should be in the Running state.

14. Check if the Database tododb Exists in the MySQL Container

1. Connect to the MySQL container:

```
bash
kubectl exec -it <mysql-pod-name> -- mysql -u root -p
```

Enter the password Root@123 when prompted.

2. List all databases:

```
sql
SHOW DATABASES;
```

Expected Output: The tododb database should be listed.

Summary

By following the above steps, you can verify all the tasks in your assessment. Here's a quick recap of the commands:

Task	Command
Check if jenkins is in the docker group	groups jenkins
Check if todo-application-image exists	docker images
Check if todo-application-image is tagged as latest	docker images grep todo-application-image`
Check if mysql-data volume exists	docker volume ls

Task	Command	
Check if todo-network exists	<code>docker network ls</code>	
Check if the application container is active	<code>docker ps</code>	
Check if mysql-db-1 container exists	<code>`docker ps</code>	<code>grep mysql-db-1`</code>
Check if todo-application-1 container exists	<code>`docker ps</code>	<code>grep todo-application-1`</code>
Check if my-registry-secret exists	<code>kubectrl get secrets</code>	
Check if the application is running on Minikube at port 30080	<code>minikube ip</code> and open <code>http://<minikube-ip>:30080</code> in a browser	
Check if the application serves expected content	<code>curl http://<minikube-ip>:30080</code>	
Check if the latest Jenkins build is successful	Check the Jenkins dashboard	
Check if all pods are in a Running state	<code>kubectrl get pods</code>	
Check if the tododb database exists	<code>kubectrl exec -it <mysql-pod-name> -- mysql -u root -p</code> and <code>SHOW DATABASES</code>	

This checklist ensures that all aspects of the deployment are validated, and the application is running as expected.

[New chat](#)