

# PropPulse CI/CD deployment (GitHub, Jenkins, DockerHub & Minikube)

<b>Version</b>	1.0.1
<b>Prepared by</b>	Rimah Houssameldine
<b>Audience</b>	ParkInnovation Team
<b>Date</b>	23/02/2024

## Table of Contents

<b>A. Introduction:</b> .....	2
1. Requirements: .....	2
<b>B. Steps for deploying PopPulse</b> .....	3
1. Create GitHub Repository .....	3
2. Push local files to the created repository.....	4
3. Create a DockerHub repository .....	5
4. Build docker images .....	6
5. Build a pipeline on jenkins.....	8
6. Add the required credentials .....	9
7. Create DB, FE & BE docker files .....	11
8. Create a database directory .....	13
9. Jenkins Pipeline execution .....	14
10. Apply Kubernetes Configuration for Account Resources .....	15
11. Creating a Production Environment.....	19

## A. Introduction:

This guide walks through deploying a PHP application using GitHub, DockerHub, Jenkins, and Minikube. It covers setting up a GitHub repository, configuring DockerHub, implementing Jenkins for automation, and utilizing Minikube for local Kubernetes deployment. By following this documentation, users can establish a seamless CI/CD pipeline for efficient development, testing, and deployment of their PHP application.

### 1. Requirements:

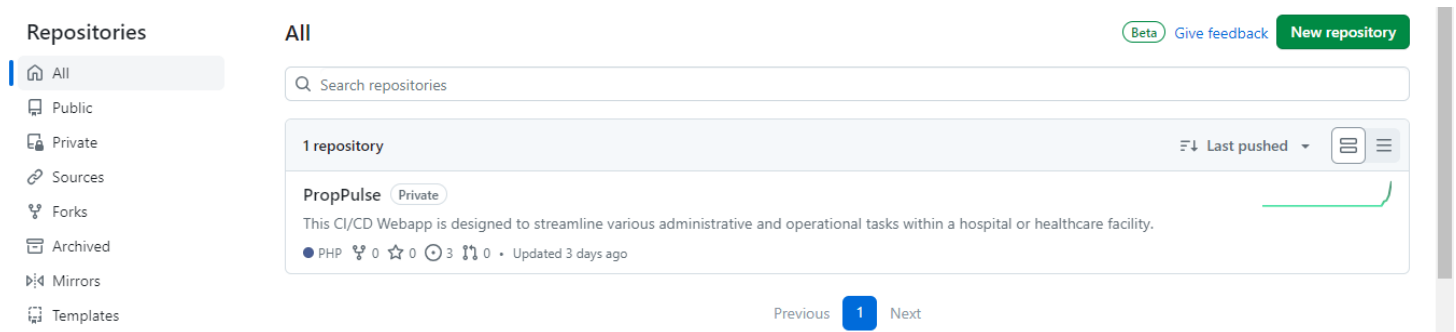
- PC (Desktop, Laptop)
- Git, GitHub, Docker engine, DockerHub, Jenkins & Minikube
- VS Code
- Browser and Internet



## B. Steps for deploying PopPulse

### 1. Create GitHub Repository

To create a new GitHub repository, navigate to your GitHub account, click on the "+" icon in the top-right corner, and select "New repository", then follow the prompts to configure repository settings and create it.



## 2. Push local files to the created repository

To push local files to a GitHub repository, first ensure you have initialized a Git repository locally using **git init**. Then, add your files using **git add .**, commit them using **git commit -m "Your commit message"**, and finally, push them to the GitHub repository using **git push origin master** (replace "master" with your branch name if applicable). Make sure you have the appropriate permissions to push to the repository and that you have configured the correct remote URL for your GitHub repository.

```
MINGW64:/c:/xampp/htdocs/Hospital-PHP

user@LAPTOP-26ATSE3V MINGW64 /c:/xampp/htdocs/Hospital-PHP (main)
$ git pull
Already up to date.

user@LAPTOP-26ATSE3V MINGW64 /c:/xampp/htdocs/Hospital-PHP (main)
$ git add .

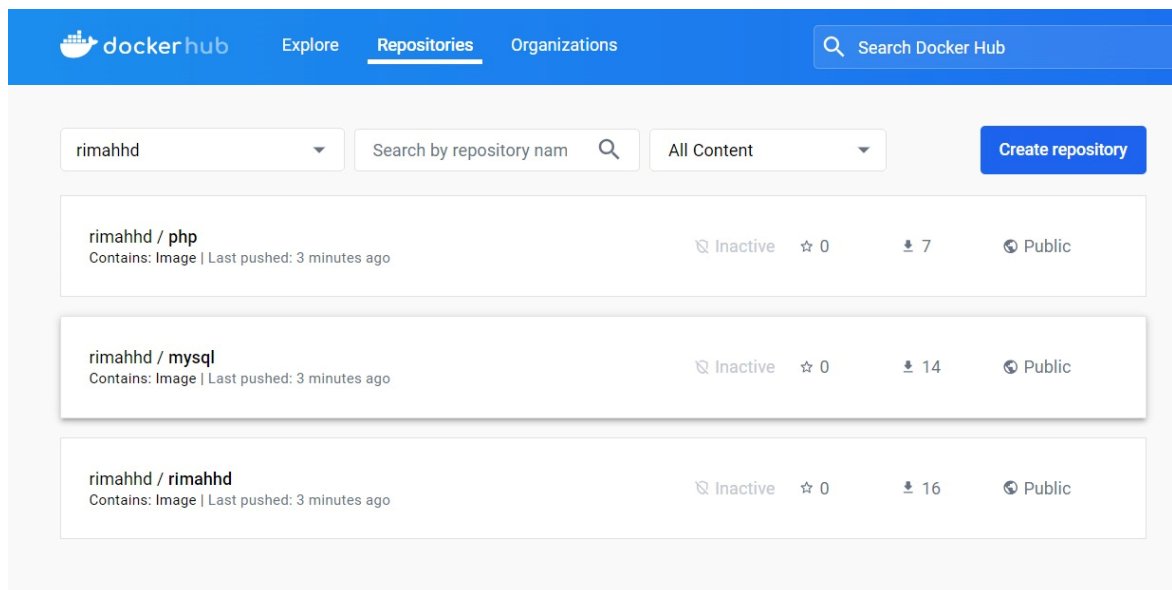
user@LAPTOP-26ATSE3V MINGW64 /c:/xampp/htdocs/Hospital-PHP (main)
$ git commit -m 'adding files'
[main 3d9eea5] adding files
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 devOps/documentations/deployment documentation/cloudway host
ing-deployment documentation/~$ploy PHP Application with Automation Tools (Githu
b&Cloudways).docx

user@LAPTOP-26ATSE3V MINGW64 /c:/xampp/htdocs/Hospital-PHP (main)
$ git push origin main
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 16 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 825 bytes | 825.00 KiB/s, done.
Total 7 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/Property-Management-System/PropPulse.git
97f527f..3d9eea5 main -> main

user@LAPTOP-26ATSE3V MINGW64 /c:/xampp/htdocs/Hospital-PHP (main)
$ |
```

### 3. Create a DockerHub repository

To create a DockerHub repository, log in to your DockerHub account, navigate to the dashboard, and click on the "Create Repository" button. Then, specify the repository name, description, visibility (public or private), and any other desired settings. Finally, click "Create" to finalize the creation of your DockerHub repository.



## 4. Build docker images

To build three Docker images for frontend, backend, and database using GitHub workflows, you'll need to set up a workflow file (e.g., **.github/workflows/docker-build.yml**) in your repository.

Within this file, define jobs for building each image using Docker build commands.

The screenshot displays the GitHub web interface for a repository named 'Property-Management-System / PropPulse'. The left sidebar shows the file explorer with the following structure:

- .github/workflows
  - build.yml**
- PropPulse webapp documentation
- assets
- backend
- devOps
- parkpulse final presentation
- reports
- Dockerfile
- Dockerfile.database
- Dockerfile.phpmyadmin
- Jenkinsfile
- README.md
- deploy.yaml
- how to run the webapp on localh...

The main content area shows the 'build.yml' file, which was updated by 'Rimahhd'. The file content is as follows:

```
1  name: Build, Push, and Deploy Docker Images
2
3  on:
4    push:
5      branches:
6        - main
7
8  jobs:
9    build:
10     runs-on: ubuntu-latest
11
12
13
14   steps:
15     - name: Checkout code
16       uses: actions/checkout@v2
17
18     - name: Set version as an environment variable
19       run: echo "VERSION=1.0.${{ github.run_number }}" >> $GITHUB_ENV
20
21     - name: Build App Docker image
22       run: docker build -t rimahhd/rimahhd:${{ env.VERSION }} .
```

yamlCopy

```
build.yml
1  name: Build, Push, and Deploy Docker Images
2
3  on:
4    push:
5      branches:
6        - main
7
8  jobs:
9    build:
10     runs-on: ubuntu-latest
11
12
13
14     steps:
15       - name: Checkout code
16         uses: actions/checkout@v2
17
18       - name: Set version as an environment variable
19         run: echo "VERSION=1.0.${{ github.run_number }}" >> $GITHUB_ENV
20
21       - name: Build App Docker image
22         run: docker build -t rimahhd/rimahhd:${{ env.VERSION }} .
23
24       - name: Build Database Docker image
25         run: docker build -t rimahhd/mysql:${{ env.VERSION }} -f Dockerfile.database .
26
27       - name: Build phpMyAdmin Docker image
28         run: docker build -t rimahhd/php:${{ env.VERSION }} -f Dockerfile.phpmyadmin .
29
30       - name: Log in to Docker
31         run: echo "${{ secrets.DOCKERHUB_TOKEN }}" | docker login -u "${{ secrets.DOCKERHUB_USERNAME }}" --password-stdin
32
33       - name: Push App Docker image
34         run: docker push rimahhd/rimahhd:${{ env.VERSION }}
35
36       - name: Push Database Docker image
37         run: docker push rimahhd/mysql:${{ env.VERSION }}
38
39       - name: Push phpMyAdmin Docker image
40         run: docker push rimahhd/php:${{ env.VERSION }}
```

Commit this workflow file to your repository, and each time you push changes to the specified branch, GitHub Actions will automatically trigger the workflow to build your Docker images. Make sure to authenticate with DockerHub in your workflow if your images are pushed to DockerHub.

Note on how to create secrets:

To create GitHub secrets, go to your repository's settings, navigate to the "Secrets" section, and click "New repository secret", then add the secret name and value, and save it.

The screenshot shows the GitHub repository settings page for 'Actions secrets and variables'. The left sidebar contains a navigation menu with sections: General, Access, Code and automation, Security, and Secrets and variables. The 'Secrets and variables' section is expanded, showing 'Actions' as the selected option. The main content area is titled 'Actions secrets and variables' and includes a description of secrets and variables, tabs for 'Secrets' and 'Variables', and a 'New repository secret' button. Below this is a table of repository secrets with columns for Name, Last updated, and actions (edit/delete). The table lists two secrets: DOCKERHUB\_TOKEN and DOCKERHUB\_USERNAME, both updated 'last week'. Below the table is a section for 'Organization secrets' which states 'There are no organization secrets available to this repository.'

Name	Last updated	
DOCKERHUB_TOKEN	last week	
DOCKERHUB_USERNAME	last week	

## 5. Build a pipeline on jenkins


To build a Jenkins pipeline, follow these steps:


1. Install Jenkins
2. Install Pipeline plugins
3. Create a Jenkinsfile
4. Define stages and steps
5. Configure Jenkins job
6. Run the job
7. Monitor pipeline execution
8. Iterate and improve





**Enter an item name**

» Required field

**Freestyle project**  
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Pipeline**  
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**Multi-configuration project**  
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

**Folder**  
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

OK

Cancel

Apply Pipeline

## 6. Add the required credentials

To add credentials in Jenkins:

Dashboard > Jenkins Pipeline1 > Configuration

**Configure**

General

Advanced Project Options

Pipeline

Definition

Pipeline script

Script ?

```
1 User
2 pipeline{
3   agent any
4   stages {
5
6
7   stage('Deploy App on k8s') {
8     steps {
9       withCredentials([
10        string(credentialsId: 'my_kubernetes', variable: 'api_token')
11      ]) {
12        bat 'kubectl --token $api_token --server http://127.0.0.1:8001 --insecure-skip-tls-verify=true apply -f deploy.yaml '
13      }
14    }
15  }
16 }
17 }
```

try sample Pipeline...

☒ Use Groovy Sandbox ?

Pipeline Syntax

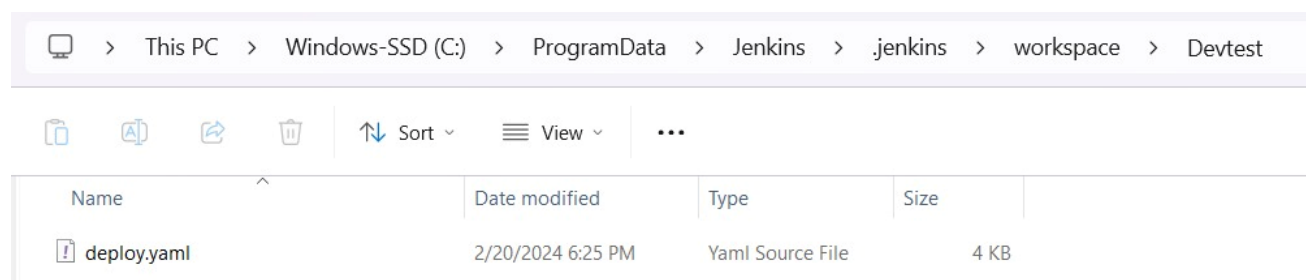
Save

Apply

1. Navigate to the Jenkins dashboard and click on "Manage Jenkins".
2. Select "Manage Credentials" from the options.
3. Click on the "Global credentials" domain (or any other domain where you want to store your credentials).

4. Click on "Add Credentials" on the left sidebar.
5. Choose the type of credential you want to add (e.g., Username with password, SSH username with private key, Secret text, etc.).
6. Fill in the required fields with your credential information.
7. Optionally, provide an ID and description for the credential.
8. Click "OK" to save the credential.

Finally, add a deploy.yaml file



The provided YAML file defines Kubernetes resources for deploying a web application, MySQL database, and PHPMyAdmin:

- **Web Application:**
  - Exposed through a NodePort service on port 80.
  - Deployed using a single replica Deployment.
- **MySQL Database:**
  - Exposed internally using a ClusterIP service on port 3306.
  - Managed by a StatefulSet for persistent storage.
- **PHPMyAdmin:**
  - Exposed through a NodePort service on port 8080.
  - Deployed using a single replica Deployment to manage the MySQL database.

These resources are configured with appropriate labels, ports, and environment variables for inter-component communication and functionality. Adjustments may be necessary based on specific deployment requirements and configurations.

## 7. Create DB, FE & BE docker files

```
Dockerfile.database > ...
1  # Use an official MySQL runtime as a parent image
2  FROM mysql:latest
3
4  # Set the MySQL root password
5  ENV MYSQL_ROOT_PASSWORD=root
6
7  # Create a database and user
8  ENV MYSQL_DATABASE=hmisphp
9  ENV MYSQL_USER=root
10  ENV MYSQL_PASSWORD=
11
12  # When container starts, execute the following SQL script
13  COPY ./init.sql /docker-entrypoint-initdb.d/
14
```

```
Dockerfile.phpmyadmin > ...
1  # Use an official phpMyAdmin image as base
2  FROM phpmyadmin/phpmyadmin:latest
3
4  # Set environment variables for MySQL connection
5  ENV PMA_HOST=localhost
6  ENV PMA_USER=root
7  ENV PMA_PASSWORD=
8
9  # The port phpMyAdmin will run on
10  EXPOSE 80
11
12  # Start phpMyAdmin
13  CMD ["apache2-foreground"]
14
```

```
Dockerfile > ...
1  # Use an existing Apache image with PHP support as a base
2  FROM php:apache
3
4  # Install the MySQLi extension
5  RUN docker-php-ext-install mysqli
6
7  # Set the working directory in the container
8  WORKDIR /var/www/html
9
10  # Copy HTML and PHP files from the host into the container
11  COPY . .
12
13  # Expose port 80 to the outside world
14  EXPOSE 80
15
```

## Start Minikube

To start Minikube using Docker engine, follow these steps:

1. **Install Docker:** If you haven't already, download and install Docker from the official Docker website according to your operating system.
2. **Install Minikube:** Download and install Minikube by following the instructions provided in the Minikube documentation, ensuring that you have the latest version installed.
3. **Start Minikube:** Open a terminal or command prompt and run the following command to start Minikube with the Docker driver:

sqlCopy code

start

This command initializes Minikube with Docker as the driver for managing the Kubernetes cluster.

4. **Verify Minikube:** Once Minikube has started successfully, verify its status by running:

luaCopy code

status

You should see the status of Minikube as "Running".

5. **Manage your Kubernetes cluster:** You can now use Minikube to manage your Kubernetes cluster locally. You can interact with the cluster using kubectl, the Kubernetes command-line tool.

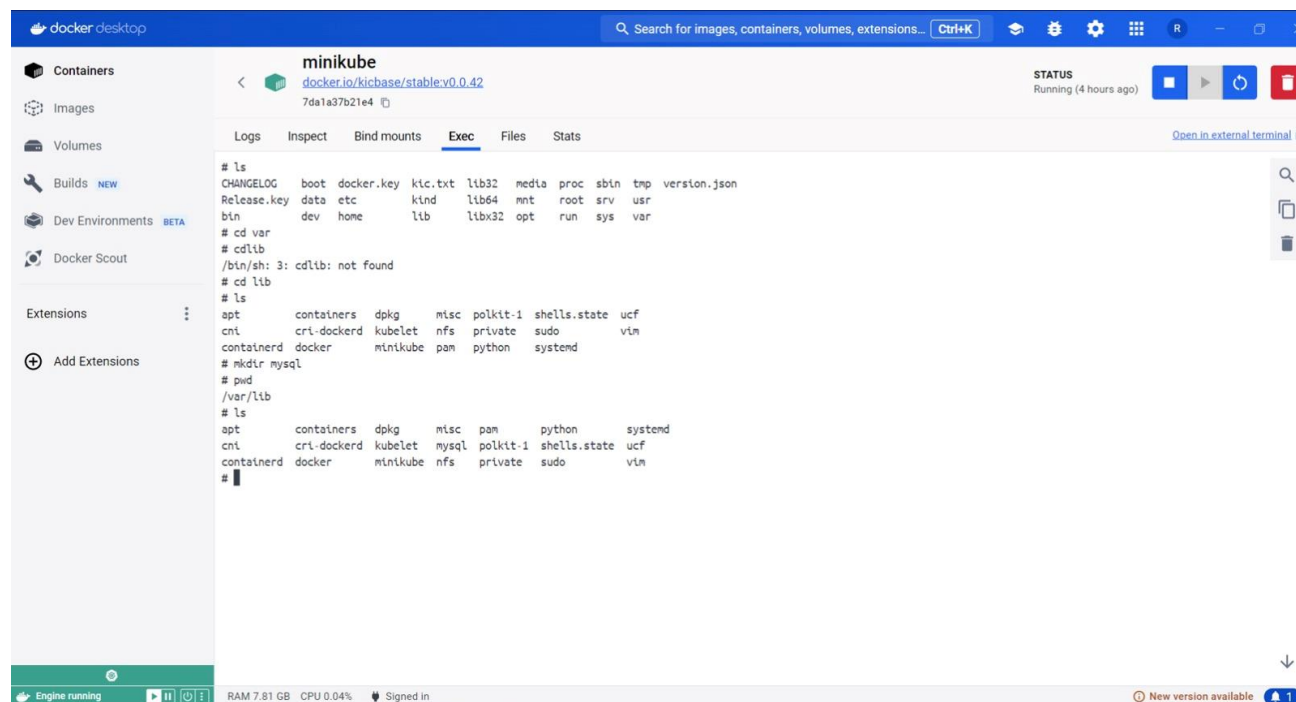
By following these steps, you can start Minikube using Docker as its driver, enabling you to run a Kubernetes cluster locally for development and testing purposes.

```
user@LAPTOP-26ATSE3V MINGW64 ~  
$ minikube start  
W0216 15:25:06.576217 24192 main.go:291] Unable to resolve the current Docker  
CLI context "default": context "default": context not found: open C:\Users\user\  
.docker\contexts\meta\37a8eec1ce19687d132fe29051dca629d164e2c4958ba141d5f4133a33  
f0688f\meta.json: The system cannot find the path specified.  
* minikube v1.32.0 on Microsoft Windows 11 Home Single Language 10.0.22631.3155  
Build 22631.3155  
* Using the docker driver based on existing profile  
* Starting control plane node minikube in cluster minikube  
* Pulling base image ...  
* Restarting existing docker container for "minikube" ...  
* Preparing Kubernetes v1.28.3 on Docker 24.0.7 ...  
* Configuring bridge CNI (Container Networking Interface) ...  
* Verifying Kubernetes components...  
  - Using image docker.io/kubernetes/dashboard:v2.7.0  
  - Using image docker.io/kubernetes/metrics-scraper:v1.0.8  
  - Using image gcr.io/k8s-minikube/storage-provisioner:v5  
* Some dashboard features require the metrics-server addon. To enable all featur  
es please run:  
  
    minikube addons enable metrics-server  
  
* Enabled addons: storage-provisioner, default-storageclass, dashboard  
* Done! kubectl is now configured to use "minikube" cluster and "default" namesp  
ace by default  
  
user@LAPTOP-26ATSE3V MINGW64 ~  
$ minikube dashboard  
W0216 15:25:48.874495 2700 main.go:291] Unable to resolve the current Docker  
CLI context "default": context "default": context not found: open C:\Users\user\  
.docker\contexts\meta\37a8eec1ce19687d132fe29051dca629d164e2c4958ba141d5f4133a33  
f0688f\meta.json: The system cannot find the path specified.  
* Verifying dashboard health ...  
* Launching proxy ...  
* Verifying proxy health ...  
* Opening http://127.0.0.1:53270/api/v1/namespaces/kubernetes-dashboard/services  
/http:kubernetes-dashboard:/proxy/ in your default browser...
```

Here Docker is taken by default, so no need to mention –driver in the CLI

## 8. Create a database directory

- Navigate to minikube container on docker desktop
- Navigate to exec



- Create a mysql directory, as appears in the above image

## 9. Jenkins Pipeline execution





```
metadata:  
  name: jenkins  
  namespace: default  
---
```

```
kind: Role  
apiVersion: rbac.authorization.k8s.io/v1  
metadata:  
  name: jenkins  
  namespace: default  
rules:  
- apiGroups: [""]  
  resources: ["pods", "services"]  
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]  
- apiGroups: ["apps"]  
  resources: ["deployments"]  
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]  
- apiGroups: [""]  
  resources: ["pods/exec"]  
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]  
- apiGroups: [""]  
  resources: ["pods/log"]  
  verbs: ["get", "list", "watch"]  
- apiGroups: [""]  
  resources: ["secrets"]  
  verbs: ["get"]  
- apiGroups: [""]  
  resources: ["persistentvolumeclaims"]  
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]  
  
---
```

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: jenkins-token  
  annotations:  
    kubernetes.io/service-account.name: jenkins  
type: kubernetes.io/service-account-token  
  
---
```


```
apiVersion: rbac.authorization.k8s.io/v1  
kind: RoleBinding  
metadata:  
  name: jenkins  
  namespace: default  
roleRef:  
  apiGroup: rbac.authorization.k8s.io  
  kind: Role  
  name: jenkins
```



```
subjects:
- kind: ServiceAccount
  name: jenkins
---
# Allows jenkins to create persistent volumes
# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: jenkins-crb
subjects:
- kind: ServiceAccount
  namespace: default
  name: jenkins
roleRef:
  kind: ClusterRole
  name: jenkinsclusterrole
  apiGroup: rbac.authorization.k8s.io
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: jenkinsclusterrole
rules:
- apiGroups: [""]
  resources: ["persistentvolumes"]
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]
```

These Kubernetes manifests define resources and permissions necessary for Jenkins to interact with the Kubernetes cluster. It includes a ServiceAccount, Role, Secret, RoleBinding, ClusterRoleBinding, and ClusterRole. These resources grant Jenkins appropriate permissions for managing pods, services, deployments, secrets, and persistent volume claims within the default namespace. Additionally, it allows Jenkins to create and manage persistent volumes across the entire cluster.

Then run `kubectrl proxy` that initiates a proxy server, facilitating a connection between your local system and the Kubernetes API server. This enables you to interact with the Kubernetes API directly from your local machine, enhancing accessibility and ease of management.

 **Jenkins**

Dashboard > Devtest >

Status

</> Changes

► Build Now

⚙️ Configure

🗑️ Delete Pipeline

🔍 Full Stage View

✎️ Rename

❓ Pipeline Syntax

Build History

trend ▼

Filter builds...

#35  
Feb 20, 2024, 6:29 PM

#34  
Feb 20, 2024, 6:28 PM

#33

✓ Devtest

Stage View

Average stage times:  
(Average full run time: ~1s)

Deploy App on k8s

679ms

#35  
Feb 20 18:29  
No Changes

643ms

#34  
Feb 20 18:28  
No Changes

653ms

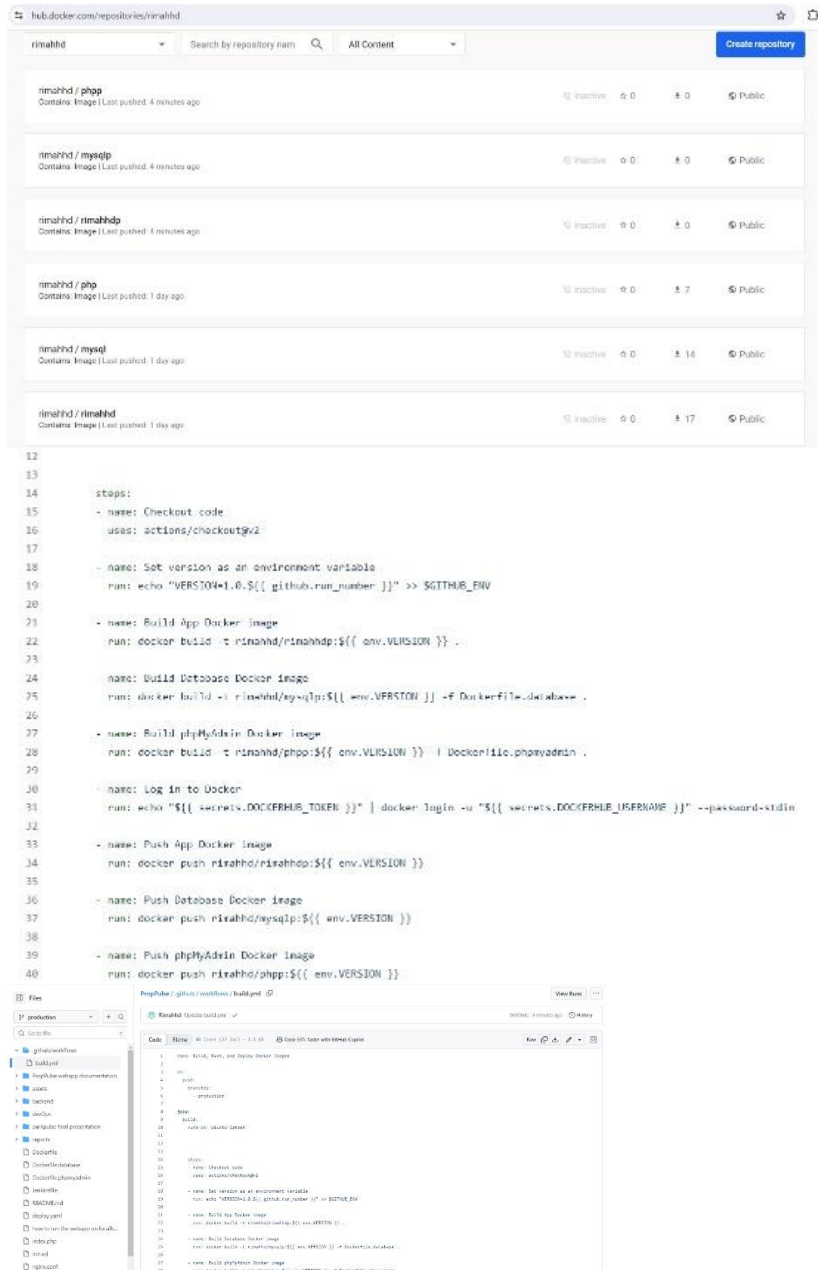
#33  
Feb 20 17:59  
No Changes

631ms

Then wait till the pods finish running

```
user@LAPTOP-26ATSE3V MINGW64 ~
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
rimahhd-phpmyadmin-deployment-645bdd4b5d-9btw5   1/1     Running   0          8s
rimahhd-web-deployment-79c669c99d-9c5d9          1/1     Running   0          8s
wissam-db-statefulset-0                         1/1     Running   0          8s
```

## 11. Creating a Production Environment



The screenshot displays the Docker Hub repository page for 'rimahhd' and a corresponding GitHub Actions workflow file. The repository page lists several Docker images: 'rimahhd/php', 'rimahhd/mysql', 'rimahhd/rimahhd', 'rimahhd/php', 'rimahhd/mysql', and 'rimahhd/rimahhd'. Below the repository page, the GitHub Actions workflow file is shown, detailing the steps for building and pushing Docker images to Docker Hub.

```
12
13
14 steps:
15   - name: Checkout code
16     uses: actions/checkout@v2
17
18   - name: Set version as an environment variable
19     run: echo "VERSION=1.0.${{ github.run_number }}" >> $GITHUB_ENV
20
21   - name: Build App Docker image
22     run: docker build -t rimahhd/rimahhd:${{ env.VERSION }} .
23
24   - name: Build Database Docker image
25     run: docker build -t rimahhd/mysql:${{ env.VERSION }} -f Dockerfile-database .
26
27   - name: Build phpMyAdmin Docker image
28     run: docker build -t rimahhd/phpmyadmin:${{ env.VERSION }} -f Dockerfile-phpmyadmin .
29
30   - name: Log in to Docker
31     run: echo "${{ secrets.DOCKERHUB_TOKEN }}" | docker login -u "${{ secrets.DOCKERHUB_USERNAME }}" --password-stdin
32
33   - name: Push App Docker image
34     run: docker push rimahhd/rimahhd:${{ env.VERSION }}
35
36   - name: Push Database Docker image
37     run: docker push rimahhd/mysql:${{ env.VERSION }}
38
39   - name: Push phpMyAdmin Docker image
40     run: docker push rimahhd/phpmyadmin:${{ env.VERSION }}
```

- Creating a production branch

S	W	Name	Last Success	Last Failure	Last Duration
		Devtest	2 days 21 hr #37	10 days #11	17 sec
		initit	N/A	N/A	N/A
		Jenkins Pipeline	N/A	1 mo 25 days #19	18 sec
		Jenkins Pipeline1	N/A	N/A	N/A
		production	1 day 22 hr #2	1 day 22 hr #1	12 sec
		projetfinal	N/A	10 days #2	12 sec
		Rimah Houssameldine	N/A	1 mo 24 days #6	27 sec
		test	1 mo 10 days #8	N/A	0.56 sec

```

43 ports:
44   - protocol: TCP
45     port: 3306
46     targetPort: 3306
47   type: ClusterIP # Change to ClusterIP, as you don't need external access
48 ---
49 apiVersion: v1
50 kind: PersistentVolume
51 metadata:
52   name: mysql-pv
53 spec:
54   capacity:
55     storage: 10Gi # Adjust the storage capacity as needed
56   volumeMode: Filesystem
57   accessModes:
58     - ReadWriteOnce
59   persistentVolumeReclaimPolicy: Retain
60   storageClassName: standard # Update with the appropriate storage class name
61   hostPath:
62     path: /var/lib/mysql # Update with the appropriate host path
63
64 ---
65
66 apiVersion: v1
67 kind: PersistentVolumeClaim
68 metadata:
69   name: database-pvc
70 spec:
71   accessModes:
72     - ReadWriteOnce
73   resources:
74     requests:
75       storage: 1Gi
76   # Reference the previously defined PersistentVolume by name
77   volumeName: mysql-pv
78
79 ---
80 apiVersion: apps/v1
81 kind: StatefulSet
82 metadata:
83   name: rimahdb statefulset
84 spec:
85   replicas: 1
86   serviceName: rimahdb.svc

```