Lab 4: Dockerize an App and Automate Image Build and Push with GitHub Actions

Lab overview

In this lab, you will explore containerizing applications using Docker and automating the build and deployment process with GitHub Actions. You'll learn Docker fundamentals, create a Dockerfile following best practices, build and run containers locally, push images to DockerHub, and set up automated workflows. By the end of this lab, you'll have a complete CI/CD pipeline that automatically builds and pushes Docker images when code changes are made.

In this lab, you will:

- Create the application and Dockerfile
- Build and run containers locally
- Push tagged images to DockerHub
- Automate Docker workflows with GitHub Actions
- Handle Docker login securely using secrets

Estimated completion time

60 minutes

Task 0: Background knowledge

The ***docker tag*** command ***adds a new name (tag)*** to an existing image.

- It does not duplicate the image data.
- Instead, it creates another reference (an alias) pointing to the same image ID.
- Tags are what allow you to version images and push them to registries like Docker Hub.

Format:

```
docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

- SOURCE_IMAGE:TAG is the local image you already built.
- TARGET_IMAGE:TAG is the new name (and version) you want to give it.

Example:

```
docker tag flask-docker-demo:latest YOUR_USERNAME/flask-docker-
demo:latest
```

What it does:

– Takes the local image flask-docker-demo:latest (created in task 2).

– Assigns it a new name in the format Docker Hub expects:

```
YOUR_USERNAME/flask-docker-demo:latest
```

- YOUR_USERNAME = your Docker Hub account name.
- flask-docker-demo = the repo name (must be lowercase).
- latest = the tag (convention for most current version).

Why it's needed:

Docker Hub requires repository names to include the username prefix. Without this, docker push won't know where to upload the image.

Creating an additional alias:

```
docker tag flask-docker-demo:latest YOUR_USERNAME/flask-docker-
demo:v1.0
```

What it does:

– Same source: the local image flask-docker-demo:latest.

– Creates an additional alias:

```
YOUR_USERNAME/flask-docker-demo:v1.0
```

Here the tag is v1.0, which represents a specific version of the image.

Why it's needed:

- **latest** is useful for always getting the most up-to-date build.
- **v1.0** (or **v2.1**, etc.) allows versioned releases. This is best practice in CI/CD pipelines because it ensures stability:
  - If someone pulls v1.0, they always get that exact version.
  - If they pull latest, they might get newer code.

Task 1: Creating the application and Dockerfile

In this task, you will create a simple Python Flask application and containerize it using Docker.

1. To create a folder named **Lab4** on the Desktop, right-click on the Desktop, click **New**, then select **Folder**. Name the folder **Lab4**. Open **VS Code,** click **File > Open Folder** and select the **Lab 4** folder.
2. To create a simple Flask application, inside the Lab4 folder, create a file named **app.py** and add the following code.

```
from flask import Flask

import os


app = Flask(__name__)


@app.route('/')

def hello():

    return f'<h1>Hello from Docker!</h1><p>Environment: 
{os.getenv("ENV", "development")}</p>'


@app.route('/health')

def health():

    return {'status': 'healthy', 'app': 'flask-docker-demo'}


if __name__ == '__main__':

    app.run(host='0.0.0.0', port=5000, debug=True)
```

3. Create a **requirements.txt** file to specify Python dependencies.

```
Flask==2.3.3
```

4. Create a **Dockerfile** in the root directory with the following content.

```
# Use official Python runtime as base image
FROM python:3.11-slim

# Set working directory in container
WORKDIR /app

# Copy requirements first (for better caching)
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Expose port 5000
EXPOSE 5000

# Create non-root user for security
RUN adduser --disabled-password --gecos '' appuser && \
    chown -R appuser:appuser /app
USER appuser
```

```
# Run the application

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

Explanation of Dockerfile best practices:

- FROM python:3.11-slim: uses a lightweight base image
- WORKDIR /app: sets a consistent working directory
- COPY requirements.txt first: leverages Docker layer caching
- --no-cache-dir: reduces image size by not storing pip cache
- Non-root user: improves security by running as non-root
- EXPOSE 5000: documents which port the app uses

Task 2: Building and running containers locally

In this task, you will build the Docker image and run it locally to test the containerized application.

1. Open the terminal in VS Code (Terminal > New Terminal) and ensure you're in the Lab4 directory.
2. Build the Docker image with a tag.

```
docker build -t flask-docker-demo:latest .
```

3. Verify the image was created successfully.

```
docker images | grep flask-docker-demo
```

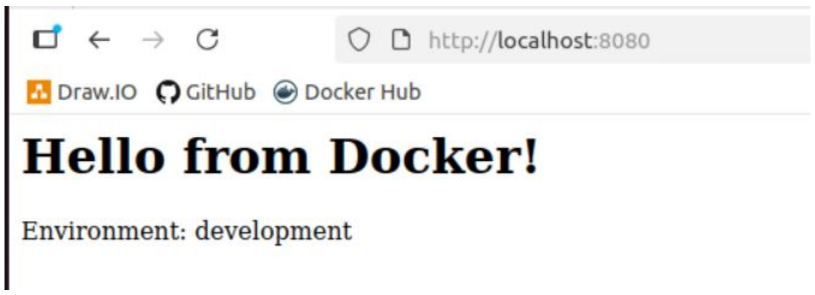
4. Run the container locally.

```
docker run -d -p 8080:5000 --name flask-app flask-docker-demo:latest
```

   Command explanation:

   – -d: run container in detached mode (background)
   – -p 8080:5000: map host port 8080 to container port 5000
   – --name flask-app: give the container a friendly name

5. Test the application by opening **http://localhost:8080** in your browser. You should see Hello from Docker! message.



Hello from Docker!

Environment: development

6. Test the health endpoint at **http://localhost:8080/health**.
7. View container logs.

```
docker logs flask-app
```

8. Stop and remove the container when done testing.

```
docker stop flask-app
```

```
docker rm flask-app
```

Task 3: Pushing images to DockerHub

In this task, you will create a DockerHub account, tag your image, and push it to the registry.

1. Create a free account at **https://hub.docker.com** if you don't have one.
2. Log in to DockerHub from your terminal.

```
docker login
```

3. Enter your DockerHub username and password when prompted.
4. Tag your image with your DockerHub username.

```
docker tag flask-docker-demo:latest YOUR_USERNAME/flask-docker-
demo:latest
```

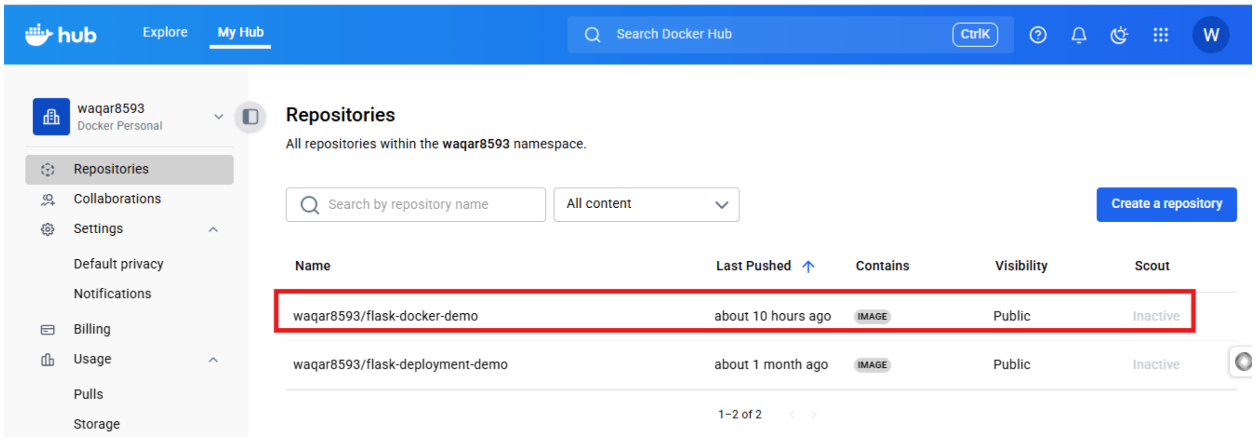
```
docker tag flask-docker-demo:latest YOUR_USERNAME/flask-docker-
demo:v1.0
```

   Replace **YOUR_USERNAME** with your actual DockerHub username.

5. Push the images to DockerHub.

```
docker push YOUR_USERNAME/flask-docker-demo:latest
docker push YOUR_USERNAME/flask-docker-demo:v1.0
```

6. Verify the image appears in your DockerHub repositories at https://hub.docker.com/repositories.



Repositories

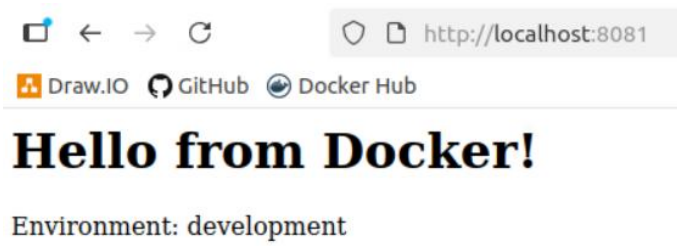
All repositories within the **waqar8593** namespace.

Name	Last Pushed	Contains	Visibility	Scout
waqar8593/flask-docker-demo	about 10 hours ago	IMAGE	Public	Inactive
waqar8593/flask-deployment-demo	about 1 month ago	IMAGE	Public	Inactive

1–2 of 2

7. Test pulling and running the image from DockerHub.

```
docker run -d -p 8081:5000 YOUR_USERNAME/flask-docker-demo:latest
```



Hello from Docker!

Environment: development

8. Stop the app.

   When you run this command:

```
docker run -d -p 8081:5000 YOUR_USERNAME/flask-docker-demo:latest
```

   it starts a detached container (-d flag) from your image, mapping port 5000 inside the container to 8081 on your host.

   To stop it, you need to identify and stop the container.

   List running containers

```
docker ps
```

   You'll see something like this:

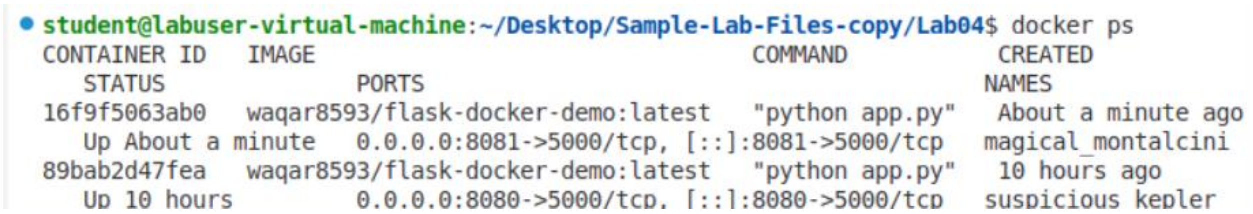
```
CONTAINER ID   IMAGE                                   COMMAND          STATUS
PORTS                        NAMES

a1b2c3d4e5f6   your_username/flask-docker-demo:latest   "python
app.py" Up 2 minutes   0.0.0.0:8081->5000/tcp   flask-demo
```

Stop the container using either its CONTAINER ID or NAMES.

```
docker stop a1b2c3d4e5f6
```

or

```
docker stop flask-demo
```



```
student@labuser-virtual-machine:~/Desktop/Sample-Lab-Files-copy/Lab04$ docker ps
CONTAINER ID   IMAGE                                 COMMAND           CREATED
   STATUS             PORTS                                          NAMES
16f9f5063ab0   waqar8593/flask-docker-demo:latest   "python app.py"   About a minute ago
   Up About a minute   0.0.0.0:8081->5000/tcp, [::]:8081->5000/tcp   magical_montalcini
89bab2d47fea   waqar8593/flask-docker-demo:latest   "python app.py"   10 hours ago
   Up 10 hours         0.0.0.0:8080->5000/tcp, [::]:8080->5000/tcp   suspicious_kepler
```

Remove the container if you don't need it anymore.

```
docker rm a1b2c3d4e5f6
```

Remove the image if you don't want it on your machine.

```
docker rmi your_username/flask-docker-demo:latest
```

Quick one-liner to stop all containers.

```
docker stop $(docker ps -q)
```

Task 4: Automating Docker builds with GitHub Actions

In this task, you will set up a GitHub Actions workflow to automatically build and push Docker images to DockerHub whenever code changes are pushed to the repository.

1. Initialize and push your code to GitHub.

```
git init
```

```
git add .
```

```
git commit -m "Initial commit: Flask Docker app"
```

2. Log in to GitHub and create a new public repository named flask-docker-demo. Link your local repo to GitHub.

```
git branch -M main
```

```
git remote add origin https://github.com/YOUR_GITHUB_USERNAME/flask-docker-demo.git
```

```
git push -u origin main
```

3. Create the workflow directory. Inside your project, create the directory for GitHub Actions workflows.

```
mkdir -p .github/workflows
```

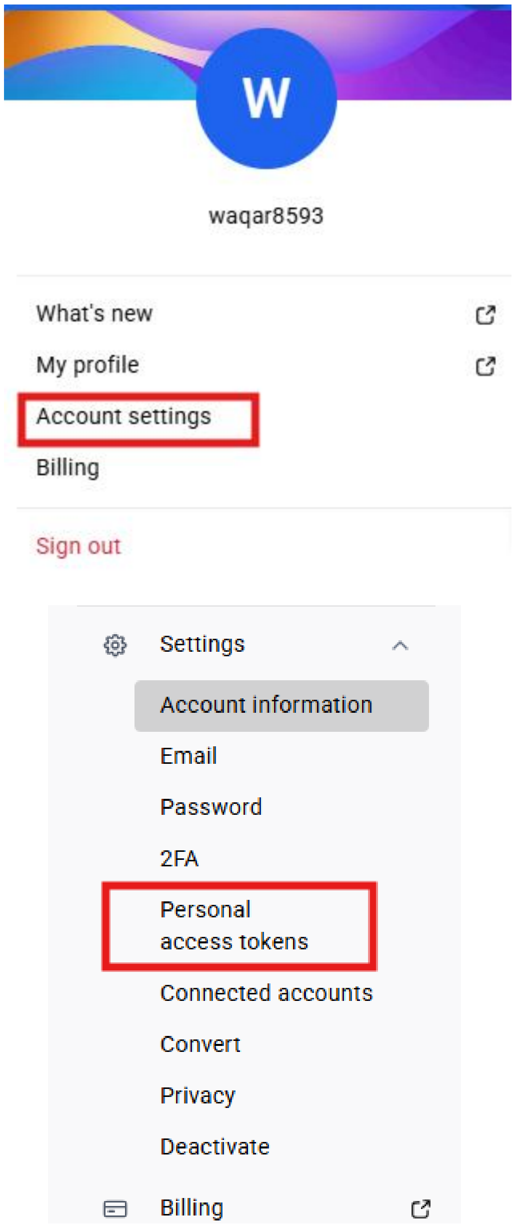
4. Create a file named .github/workflows/docker-build-push.yml with the following content.

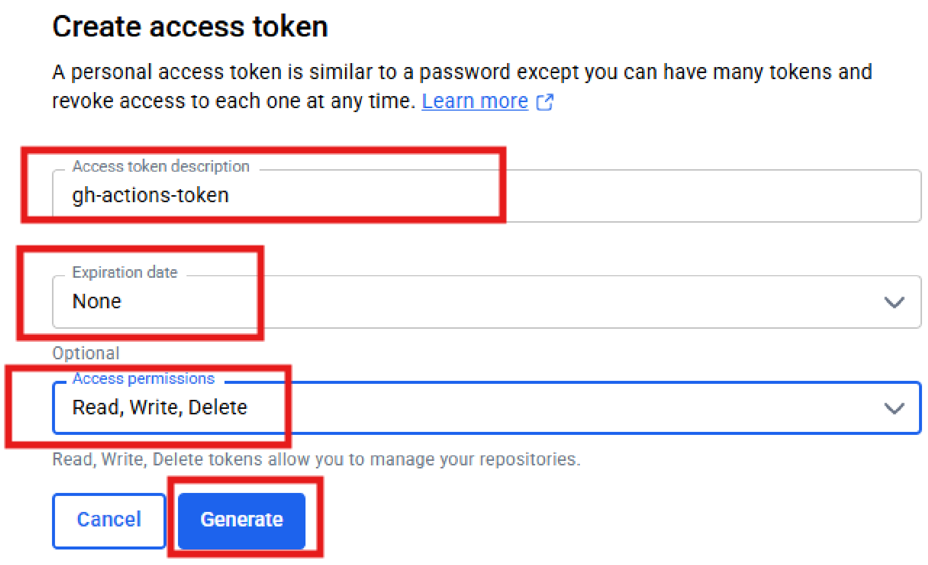
```
name:  Build and Push Docker Image


on:

  push:

    branches: [ main ]

  pull_request:

    branches: [ main ]


jobs:

  build-and-push:

    runs-on: ubuntu-latest


    steps:

    - name: Checkout code

      uses: actions/checkout@v4


    - name: Set up Docker Buildx

      uses: docker/setup-buildx-action@v3


    - name: Login to DockerHub

      if: github.event_name != 'pull_request'

      uses: docker/login-action@v3

      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}

    - name: Extract metadata
      id: meta
      uses: docker/metadata-action@v5
      with:
        images: ${{ secrets.DOCKERHUB_USERNAME }}/flask-docker-demo
        tags: |
          type=ref,event=branch
          type=ref,event=pr
          type=raw,value=latest,enable={{is_default_branch}}

    - name: Build and push Docker image
      uses: docker/build-push-action@v5
      with:
        context: .
        push: ${{ github.event_name != 'pull_request' }}
        tags: ${{ steps.meta.outputs.tags }}
        labels: ${{ steps.meta.outputs.labels }}
        cache-from: type=gha
        cache-to: type=gha,mode=max
```

5. To generate the Dockerhub personal access token (PAT), go to **DockerHub > Account Settings > Settings > Personal access tokens**. Give it a descriptive name like **gh-actions-token**.

Important

You do not use your GitHub token for DockerHub login. You must create a DockerHub token, not a GitHub one.





Create access token

A personal access token is similar to a password except you can have many tokens and revoke access to each one at any time. Learn more

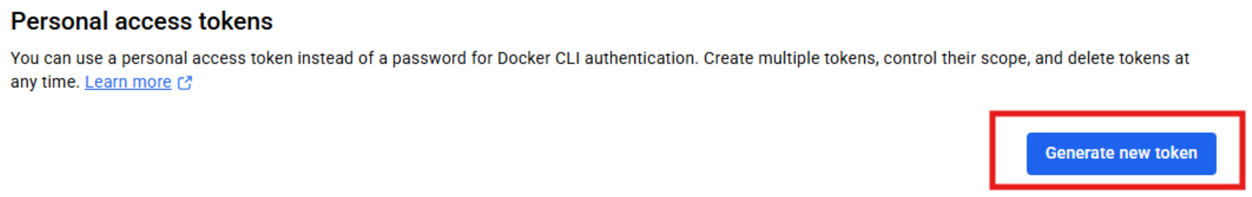
Access token description: gh-actions-token

Expiration date: None

Optional

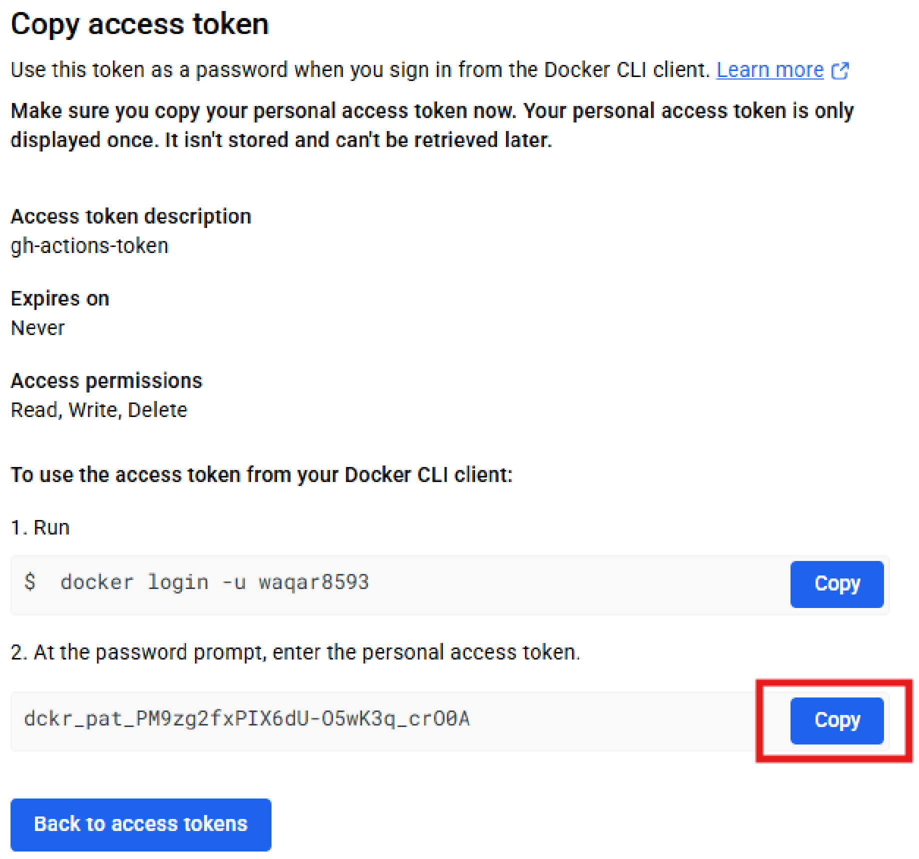
Access permissions: Read, Write, Delete

Read, Write, Delete tokens allow you to manage your repositories.



Personal access tokens

You can use a personal access token instead of a password for Docker CLI authentication. Create multiple tokens, control their scope, and delete tokens at any time. Learn more



Copy access token

Use this token as a password when you sign in from the Docker CLI client. Learn more

Make sure you copy your personal access token now. Your personal access token is only displayed once. It isn't stored and can't be retrieved later.

Access token description
gh-actions-token

Expires on
Never

Access permissions
Read, Write, Delete

To use the access token from your Docker CLI client:

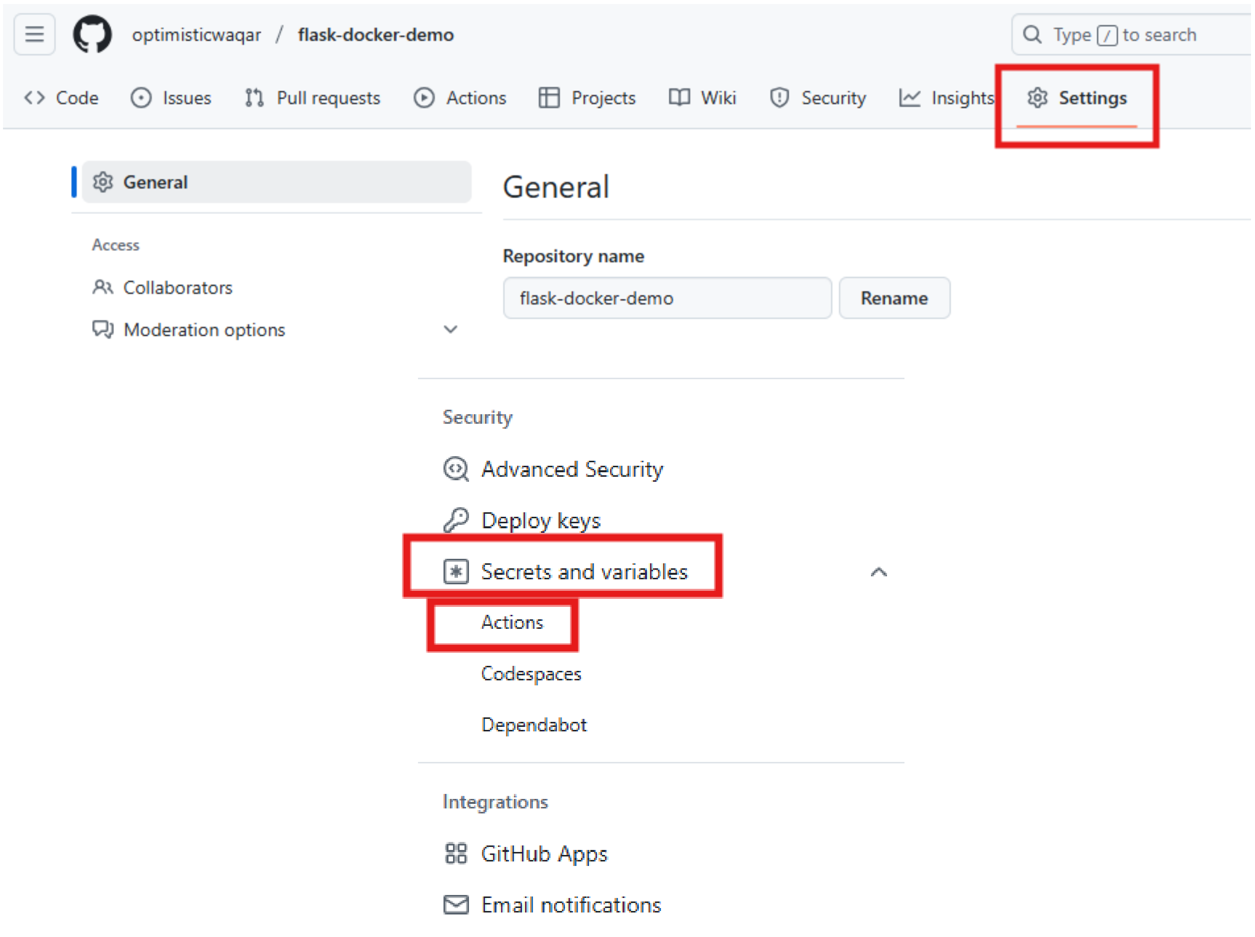
1. Run

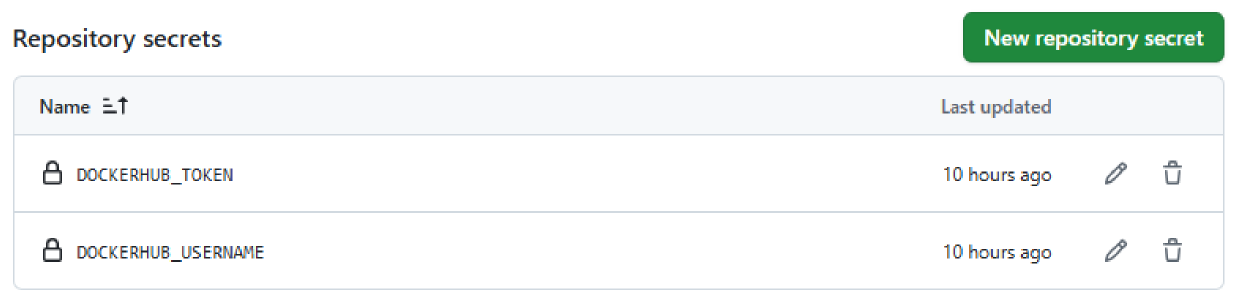
```
$  docker login -u waqar8593
```

2. At the password prompt, enter the personal access token.

```
dckr_pat_PM9zg2fxPIX6dU-O5wK3q_crO0A
```

6. To authenticate securely with DockerHub, add secrets in your GitHub repo. Go to **GitHub Repo > Settings > Secrets and variables > Actions > New repository secret**.





Name	Last updated
DOCKERHUB_TOKEN	10 hours ago
DOCKERHUB_USERNAME	10 hours ago

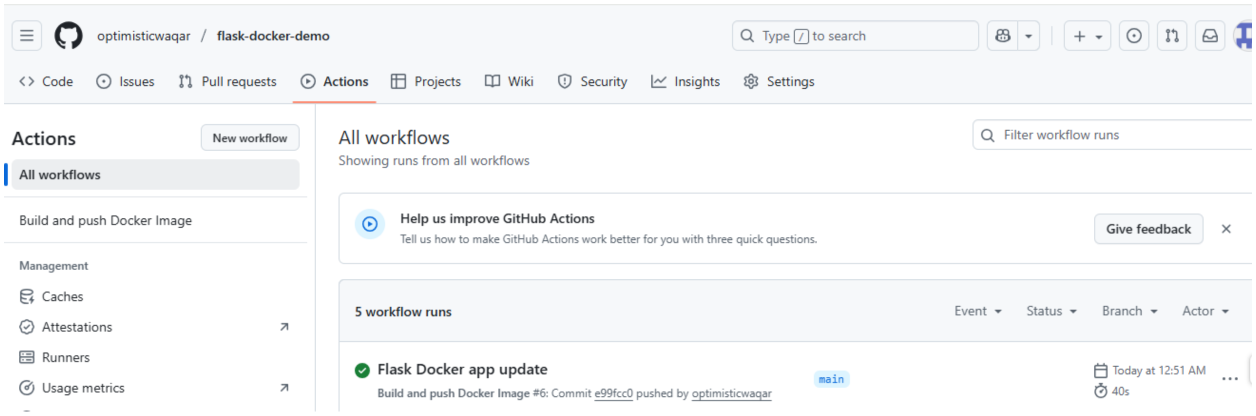
7. Add the following:
   - 1. DOCKERHUB_USERNAME > your DockerHub username
   - 2. DOCKERHUB_TOKEN > DockerHub PAT

   Use the PAT as generated above.

8. Commit and push the workflow file.

```
git add .github/
git commit -m "Add GitHub Actions workflow for Docker build & push"
git push origin main
```

9. Check the **Actions** tab in your GitHub repository to see the workflow running.



All workflows
Showing runs from all workflows

Help us improve GitHub Actions
Tell us how to make GitHub Actions work better for you with three quick questions.

5 workflow runs

Flask Docker app update
Build and push Docker Image #6: Commit e99fcc0 pushed by optimisticwaqar
Today at 12:51 AM
40s

Workflow explanation:

- **Triggers:** run on push to main branch and pull requests
- **Security:** uses secrets for DockerHub credentials
- **Caching:** utilizes GitHub Actions cache for faster builds
- **Conditional push:** only pushes on main branch, not PRs
- **Auto-tagging:** automatically tags images based on branch/PR

Task 5: Cleaning up (local and DockerHub)

When working with Docker and GitHub Actions, images and containers can accumulate over time. Cleanup ensures your system and DockerHub don't get cluttered.

1. To clean up locally (your machine), stop and remove all containers.

```
docker stop $(docker ps -q)
docker rm $(docker ps -aq)
```

2. Remove unused images.

```
docker image prune -a -f
```

3. Remove unused volumes and networks. This command deletes all volumes not currently used by a running container

```
docker volume prune -f
```

Docker volumes:

- **What they are**: volumes are special storage areas managed by Docker. They let containers store data that persists even after the container is deleted.
- **Example**: a database container may store its data in a volume.
- **Unused volumes**: if you delete the container but forget the volume, the data stays on your system. Over time, many "orphaned" volumes may pile up.

```
docker network prune -f
```

4. A full cleanup removes all networks that aren't in use by at least one container.

```
docker system prune -a -f --volumes
```

Docker networks:

- **What they are**: networks are how containers talk to each other (and to your host machine).
- By default, Docker creates a bridge network, but you can create custom networks to connect multiple containers.
- **Unused networks**: When containers are deleted, their custom networks may remain. If they're not connected to any containers, they're just taking space.

5. To clean up on DockerHub, log in to your DockerHub account.
6. Navigate to your repository (e.g., flask-docker-demo).
7. Delete old or unused tags (e.g., v1.0, dev, test).
8. Keep only latest and stable releases you want to share.

Lab review

1. What is the primary security benefit of adding a non-root user in the Dockerfile?
   A. Reduces image size
   B. Improves build performance
   C. Minimizes attack surface if container is compromised
   D. Enables better logging

STOP

You have successfully completed this lab.