Getting Started
Frequently Asked Questions
Part 1
Part 2
Part 3
Introduction to Part 3
Official Images and trust
Deployment pipelines
Using a non-root user
Optimizing the image size
Multi-host environments
End

Exercises 3.1-3.4

Deployment pipelines

CI/CD pipeline (sometimes called deployment pipeline) is a corner stone of DevOps. According to Gitlab:

CI/CD automates much or all of the manual human intervention traditionally needed to get new code from a commit into production. With a CI/CD pipeline, development teams can make changes to code that are then automatically tested and pushed out for delivery and deployment. Get CI/CD right and downtime is minimized and code releases happen faster.

Let us now see how one can set up a deployment pipeline that can be used to automatically deploy containerized software to *any* machine. So every time you commit the code in your machine, the pipeline builds the image and starts it up in the server.

Since we cannot assume that everyone has access to their own server, we will demonstrate the pipeline using *a local machine* as the development target, but the exactly same steps can be used for a virtual machine in the cloud (such as one provided by Hetzner) or even Raspberry Pi.

We will use GitHub Actions to build an image and push the image to Docker Hub, and then use a project called Watchtower to automatically pull and restart the new image in the target machine.

As an example, we will look repository https://github.com/docker-hy/docker-hy.github.io, that is, the material of this course.

As was said GitHub Actions is used to implement the first part of the deployment pipeline. The documentation gives the following overview:

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test commit and every pull request to your repository, or deploy merged pull requests to production.

The project defines a *workflow* with GitHub Actions that builds a Docker image and pushes it to Docker Hub every time the code is pushed to the GitHub repository.

Let us now see how the workflow definition looks. It is stored in the file *deploy.yml* inside the *.github/workflows* directory:

```yaml
name: Release DevOps with Docker # Name of the workflow

# On a push to the branch named master
on:
  push:
    branches:
      - master

# Job called build runs-on ubuntu-latest
jobs:
  deploy:
    name: Deploy to GitHub Pages
    # we are not interested in this job

  publish-docker-hub:
    name: Publish image to Docker Hub
    runs-on: ubuntu-latest
    steps:
    # Checkout to the repository
    - uses: actions/checkout@v2

    # We need to login so we can later push the image without issues.
    - name: Login to Docker Hub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}
    # Builds devopsdockeruh/docker-hy.github.io
    - name: Build and push
      uses: docker/build-push-action@v2
      with:
        push: true
        tags: devopsdockeruh/coursepage:latest
```

The workflow has two jobs, we are now interested in the one that is called *publish-docker-hub*. The other job, called *deploy* takes care of deploying the page as a GitHub page.

A job consists of a series of steps. Each step is a small operation or *action* that does its part of the whole. The steps are the following

- actions/checkout@v2 is used to check out the code from the repository
- docker/login-action@v1 is used to log in to Docker Hub
- docker/build-push-action@v2 is used to build the image and push it to Docker Hub

The first action was one of the ready-made actions that GitHub provides. The latter two are official actions offered by Docker. See here for more info about the official Docker GitHub Actions.

Before the workflow will work, two secrets should be added to the GitHub repository: `DOCKERHUB_TOKEN` and `DOCKERHUB_USERNAME`. This is done by opening the repository in the browser and first pressing *Settings* then *Secrets*. The `DOCKERHUB_TOKEN` can be created in Docker Hub from the *Account Settings / Security*.

GitHub Actions are doing only the "first half" of the deployment pipeline: they are ensuring that every push to GitHub is built to a Docker image which is then pushed to Docker Hub.

The other half of the deployment pipeline is implemented by a containerized service called Watchtower which is an open-source project that automates the task of updating images. Watchtower will pull the source of the image (in this case Docker Hub) for changes in the containers that are running. The container that is running will be updated and automatically restarted when a new version of the image is pushed to Docker Hub. Watchtower respects tags e.g. q container using ubuntu:22.04 will not be updated unless a new version of ubuntu:22.04 is released.

SECURITY REMINDER: DOCKER HUB ACCESSING YOUR COMPUTER

Note that now anyone with access to your Docker Hub also has access to your PC through this. If they push a malicious update to your application, Watchtower will happily download and start the updated version.

Watchtower can be run eg. using the following Docker Compose file:

```yaml
version: "3.8"

services:
  watchtower:
    image: containrrr/watchtower
    environment:
      -  WATCHTOWER_POLL_INTERVAL=60 # Poll every 60 seconds
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    container_name: watchtower
```

One needs to be careful when starting Watchtower with *docker compose up*, since it will try to update **every** image running the machine. The documentation describes how this can be prevented.

Exercises 3.1-3.4

EXERCISE 3.1: YOUR PIPELINE

Create now a similar deployment pipeline for a simple Node.js/Express app found here.

Either clone the project or copy the files to your own repository. Set up a similar deployment pipeline (or the "first half") using GitHub Actions that was just described. Ensure that a new image gets pushed to Docker Hub every time you push the code to GitHub (you may eg. change the message the app shows).

Note that there is important change that you should make to the above workflow configuration, the branch should be named *main*:

```yaml
name: Release Node.js app

on:
  push:
    branches:
```

```
    - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      # ...
```

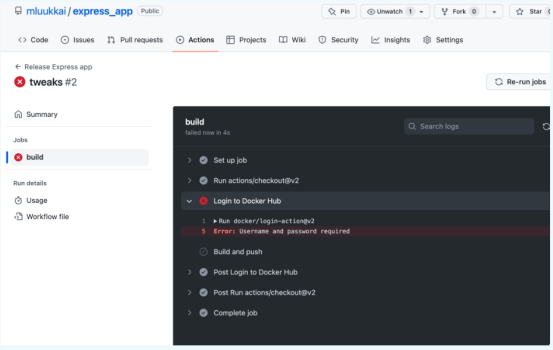
The earlier example still uses the old GitHub naming convention and calls the main branch *master*.

Some of the actions that the above example uses are a bit outdated, so go through the documentation

- actions/checkout
- docker/login-action
- docker/build-push-action

and use the most recent versions in your workflow.

Keep an eye on the GitHub Actions page to see that your workflow is working:



Ensure also from Docker Hub that your image gets pushed there.

Next, run your image locally in detached mode, and ensure that you can access it with the browser.

Now set up and run the Watchtower just as described above.

You might do these two in a single step in a shared Docker Compose file.

Now your deployment pipeline is set up! Ensure that it works:

- make a change to your code
- commit and push the changes to GitHub
- wait for some time (the time it takes for GitHub Action to build and push the image plus the Watchtower poll interval)
- reload the browser to ensure that Watchtower has started the new version (that is, your changes are visible)

Submit a link to the repository with the config.

EXERCISE 3.2: A DEPLOYMENT PIPELINE TO A CLOUD SERVICE

In Exercise 1.16 you deployed a containerized app to a cloud service.

Now it is time to improve your solution by setting up a deployment pipeline for it so that every push to GitHub results in a new deployment to the cloud service.

You will most likely find a ready-made GitHub Action that does most of the heavy lifting your you... Google is your friend!

Submit a link to the repository with the config. The repository README should have a link to the deployed application.

EXERCISE 3.3: SCRIPTING MAGIC

Create a *now* script/program that downloads a repository from GitHub, builds a Dockerfile located in the root and then publishes it into the Docker Hub.

You can use any scripting or programming language to implement the script. Using shell script might make the next exercise a bit easier... and do not worry if you have not done a shell script earlier, you do not need much for this exercise and Google helps.

The script could eg. be designed to be used so that as the first argument it gets the GitHub repository and as the second argument the Docker Hub repository. Eg. when run as follows

```
./builder.sh mluukkai/express_app mluukkai/testing
```

the script clones https://github.com/mluukkai/express_app, builds the image, and pushes it to Docker Hub repository mluukkai/testing

EXERCISE 3.4: BUILDING IMAGES FROM INSIDE OF A CONTAINER

As seen from the Docker Compose file, the Watchtower uses a volume to docker.sock socket to access the Docker daemon of the host from the container:

```
services:
watchtower:
  image: containrrr/watchtower
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
  # ...
```

In practice this means that Watchtower can run commands on Docker the same way we can "command" Docker from the cli with *docker ps*, *docker run* etc.

We can easily use the same trick in our own scripts! So if we mount the *docker.sock* socket to a container, we can use the command *docker* inside the container, just like we are using it in the host terminal!

Dockerize now the script you did for the previous exercise. You can use images from this repository to run Docker inside Docker!

Your Dockerized could be run like this (the command is divided into many lines for better readability, note that copy-pasting a multiline command does not work):

```
docker run -e DOCKER_USER=mluukkai \
  -e DOCKER_PWD=password_here \
  -v /var/run/docker.sock:/var/run/docker.sock \
  builder mluukkai/express_app mluukkai/testing
```

Note that now the Docker Hub credentials are defined as environment variables since the script needs to log in to Docker Hub for the push.

Submit the Dockerfile and the final version of your script.

Hint: you quite likely need to use ENTRYPOINT in this Exercise. See Part 1 for more.

Edit this page

Previous
« Official Images and trust

Next
Using a non-root user »