

Docker and Docker Compose

<https://github.com/heig-vd-dai-course>

[Web](#) • [PDF](#)

L. Delafontaine and H. Louis, with the help of GitHub Copilot.

This work is licensed under the [CC BY-SA 4.0](#) license.

Objectives

- Learn the differences between bare metal, virtualization and containerization
- Learn how the OCI specification defines images, containers, and registries
- Learn how to use Docker and Docker Compose to build, publish, and run applications in containers

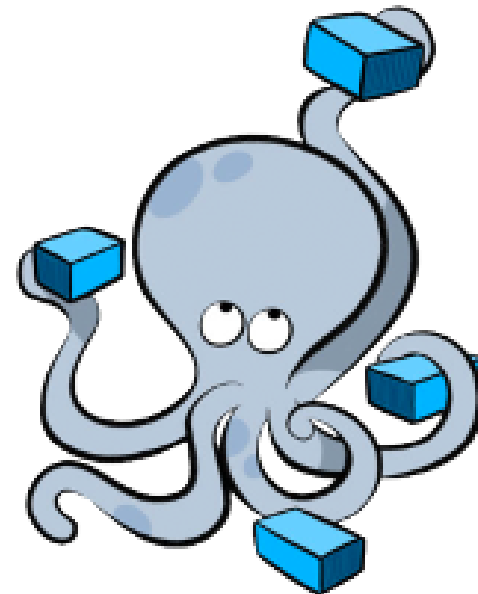


Prepare and setup your environment

More details for this section in the [course material](#). You can find other resources and alternatives as well.

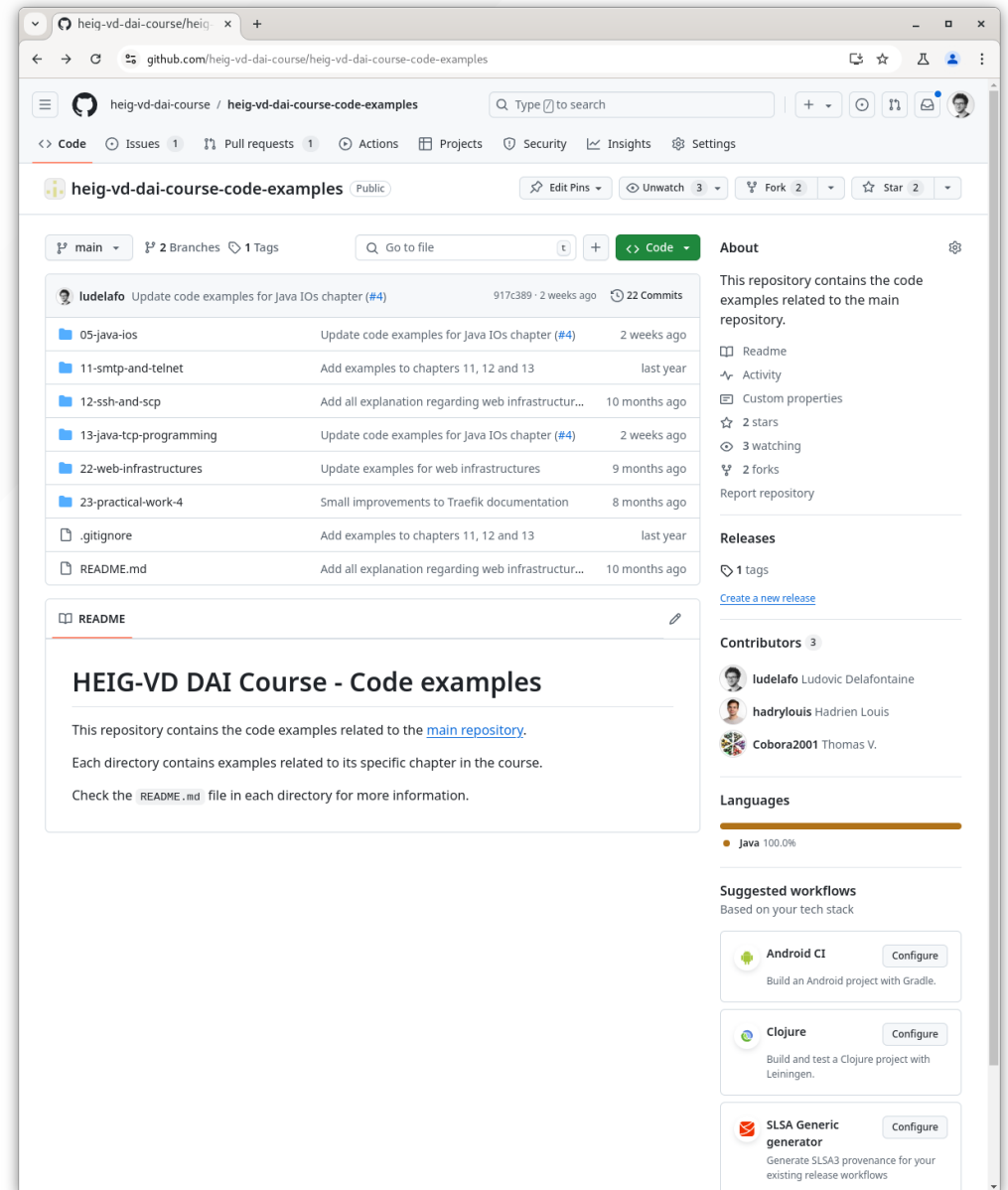
Install Docker and Docker Compose

- Install Docker and Docker Compose
- Configure Docker and Docker Compose to:
 - Run without `sudo` (root)
 - Start automatically at boot



Check and run the code examples

- Check the code examples
- Run the code examples
- Helps to understand the concepts
- Modify/play with the code examples



Bare metal, virtualization and containerization

More details for this section in the [course material](#). You can find other resources and alternatives as well.

Bare metal, virtualization and containerization

- Bare metal: software runs directly on hardware
- Virtualization: software runs on a virtual machine
- Containerization: software runs in a container

The screenshot shows a YouTube video player with the following details:

- Video Title:** Big Misconceptions about Bare Metal, Virtual Machines, and Containers
- Channel:** ByteByteGo (551K subscribers)
- Engagement:** 8.7K likes, 193K views, 1 year ago
- Thumbnail:** Docker in 1 Hour
- Video Description:** System Design Fundamentals Weekly system design newsletter: <https://bit.ly/3tfAIYD> Checkout our bestselling System Design Interview books: ...more

Bare metal

- The traditional way to run software
- Software runs directly on hardware
- Software has full access to the hardware
- Security issues, hard to maintain, hard to migrate



Virtualization

- Virtualization runs virtual machines
- A virtual machine is complete operating system
- A virtual machine is isolated from the host
- Virtual machines are heavy and use a lot of resources



Containerization

- Containerization starts containers
- Containers contain all the dependencies to run the software
- Containers are isolated from each other
- Containers are lightweight and use the host kernel

Comparing Containers and Virtual Machines

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.

CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

VIRTUAL MACHINES

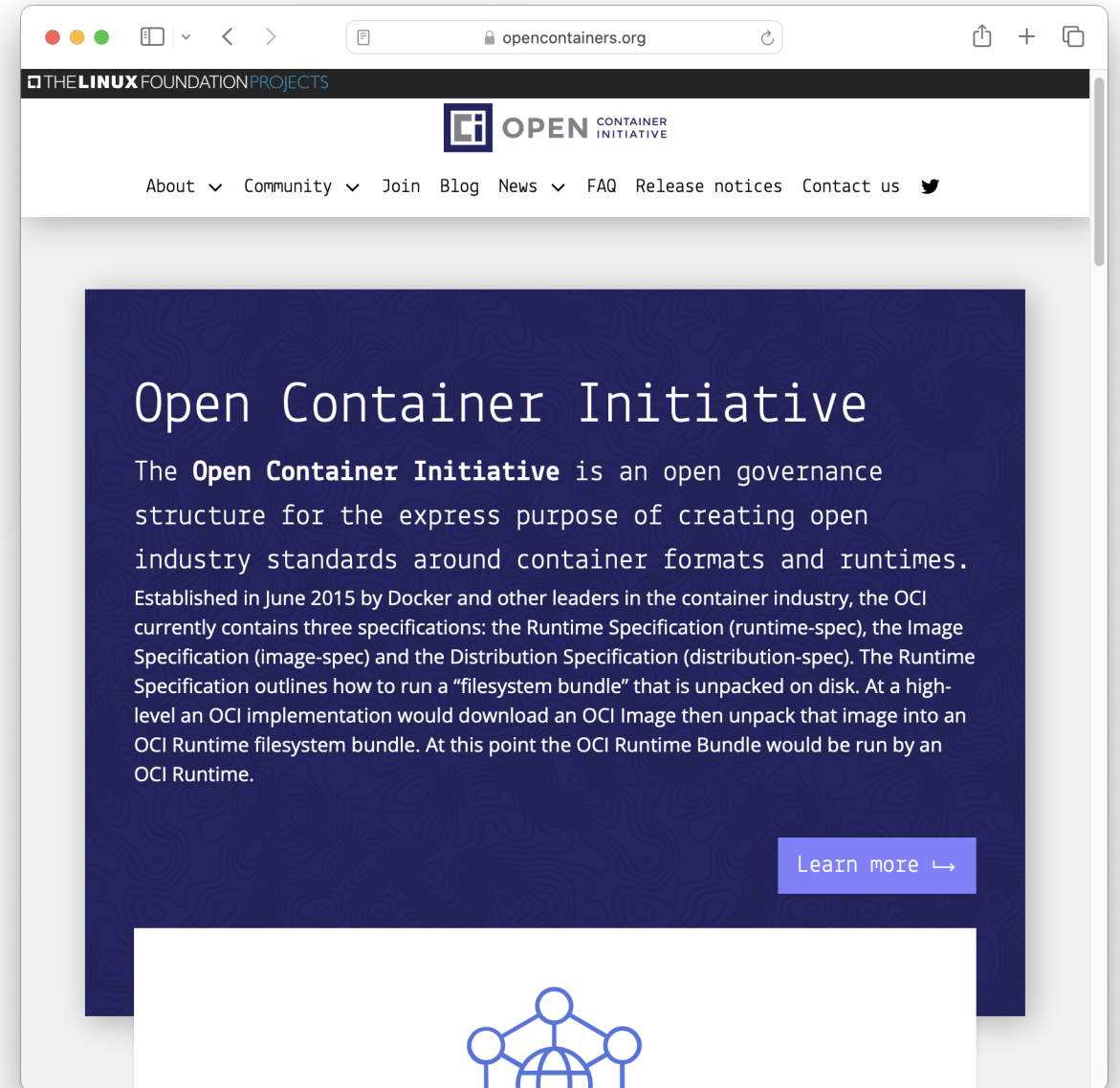
Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.

OCI, images, containers, and registries

More details for this section in the [course material](#). You can find other resources and alternatives as well.

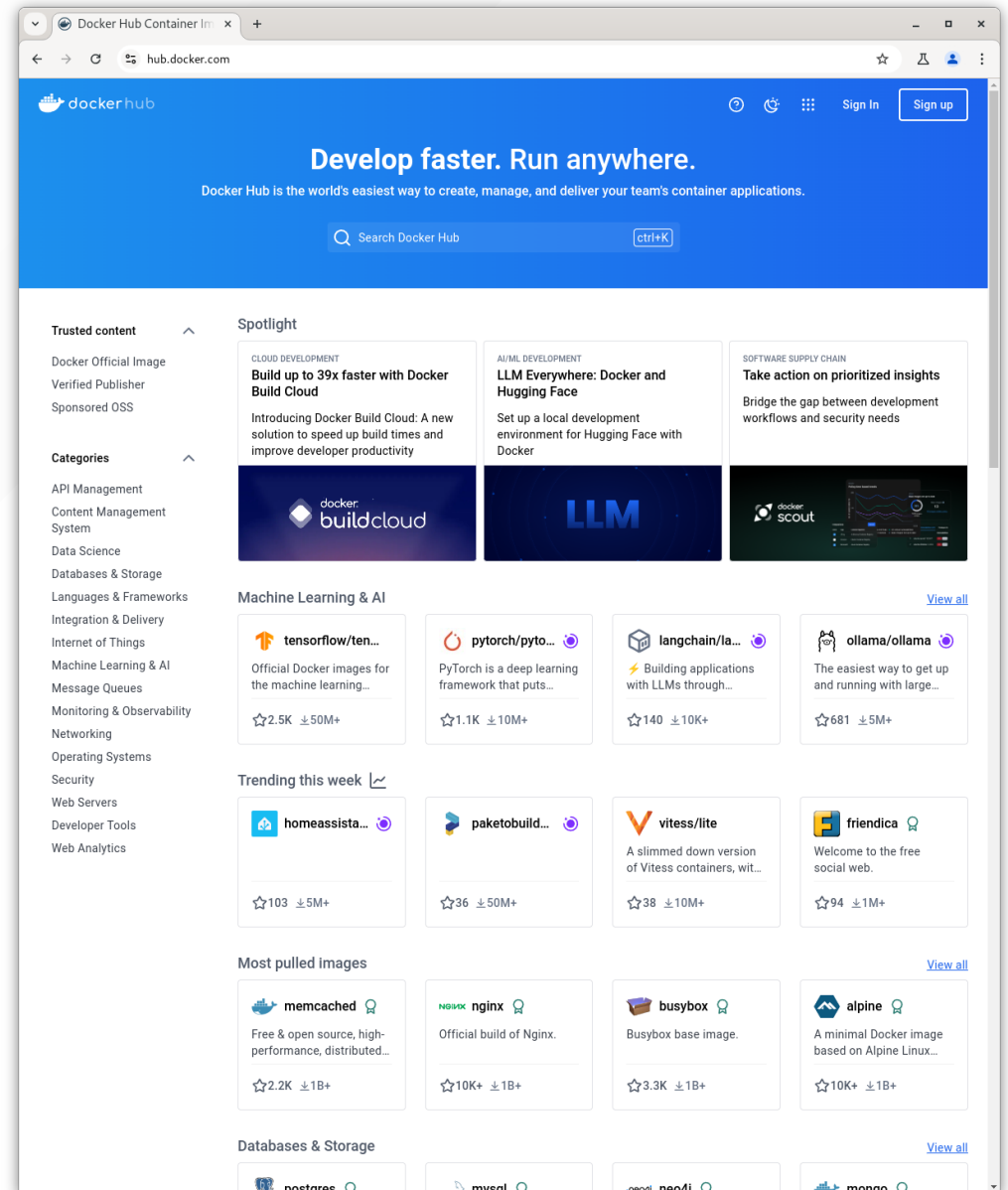
OCI, images, containers, and registries

- Image: read-only template for container creation
- Container: runnable instance of an image
- Registry: service storing images



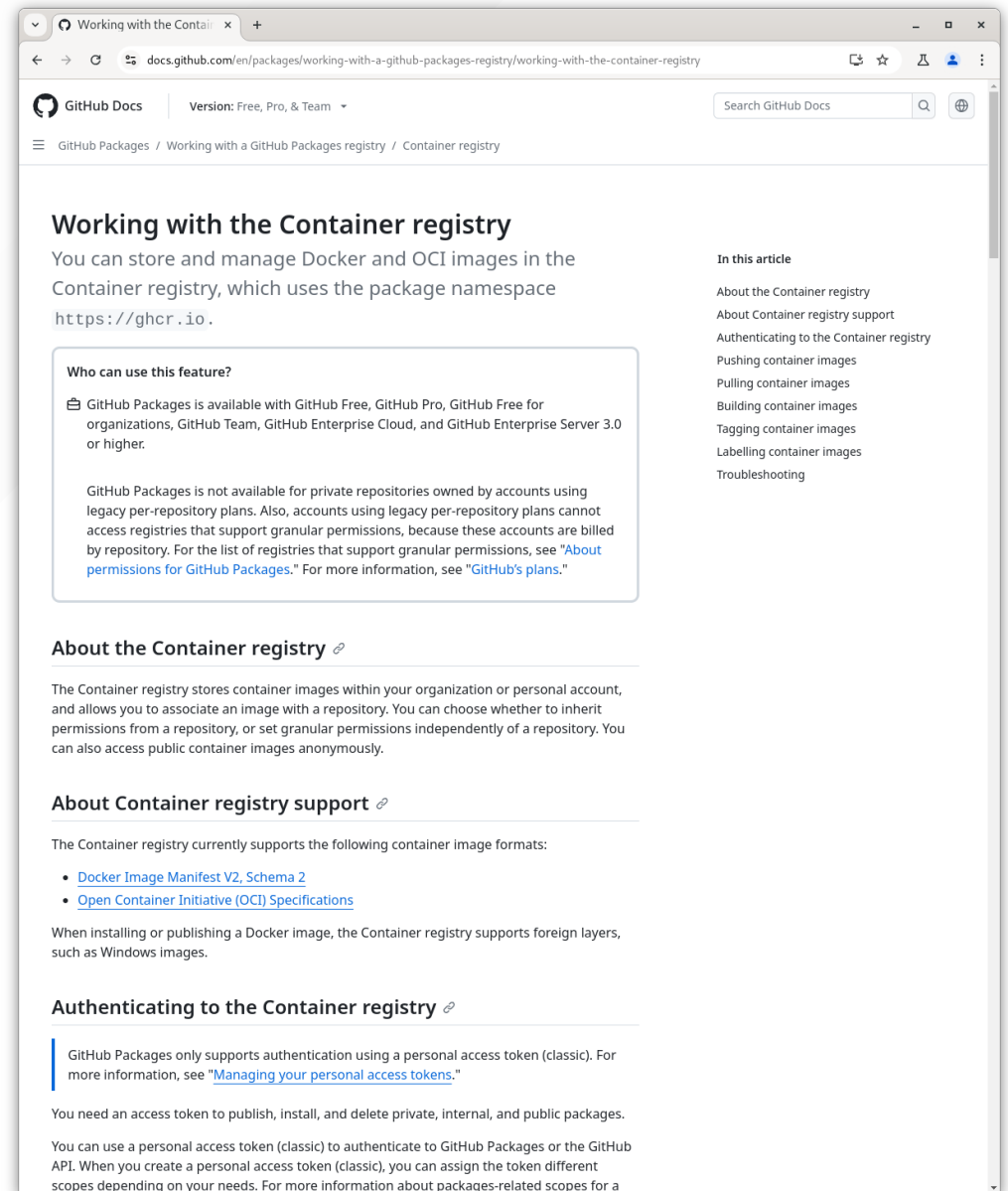
Docker Hub

- The official registry
- Hosts millions of images
- Can be used to store and share images



GitHub Container Registry

- GitHub's registry
- Hosts images in the same place as the code
- Will be used in this course for simplicity



The screenshot shows a web browser displaying the GitHub Docs page for "Working with the Container registry". The page title is "Working with the Container registry" and the subtitle is "You can store and manage Docker and OCI images in the Container registry, which uses the package namespace `https://ghcr.io`."

Who can use this feature?

GitHub Packages is available with GitHub Free, GitHub Pro, GitHub Free for organizations, GitHub Team, GitHub Enterprise Cloud, and GitHub Enterprise Server 3.0 or higher.

GitHub Packages is not available for private repositories owned by accounts using legacy per-repository plans. Also, accounts using legacy per-repository plans cannot access registries that support granular permissions, because these accounts are billed by repository. For the list of registries that support granular permissions, see "[About permissions for GitHub Packages](#)." For more information, see "[GitHub's plans](#)."

About the Container registry

The Container registry stores container images within your organization or personal account, and allows you to associate an image with a repository. You can choose whether to inherit permissions from a repository, or set granular permissions independently of a repository. You can also access public container images anonymously.

About Container registry support

The Container registry currently supports the following container image formats:

- [Docker Image Manifest V2, Schema 2](#)
- [Open Container Initiative \(OCI\) Specifications](#)

When installing or publishing a Docker image, the Container registry supports foreign layers, such as Windows images.

Authenticating to the Container registry

GitHub Packages only supports authentication using a personal access token (classic). For more information, see "[Managing your personal access tokens](#)."

You need an access token to publish, install, and delete private, internal, and public packages.

You can use a personal access token (classic) to authenticate to GitHub Packages or the GitHub API. When you create a personal access token (classic), you can assign the token different scopes depending on your needs. For more information about packages-related scopes for a

In this article

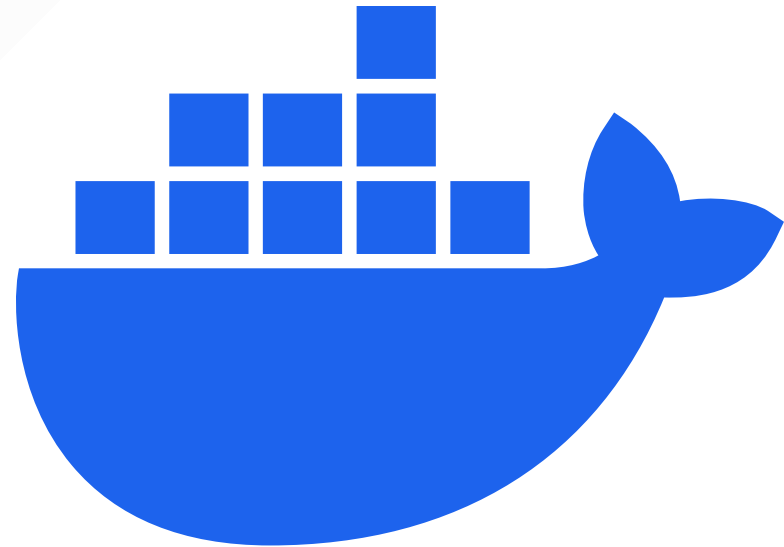
- About the Container registry
- About Container registry support
- Authenticating to the Container registry
- Pushing container images
- Pulling container images
- Building container images
- Tagging container images
- Labelling container images
- Troubleshooting

Docker

More details for this section in the [course material](#). You can find other resources and alternatives as well.

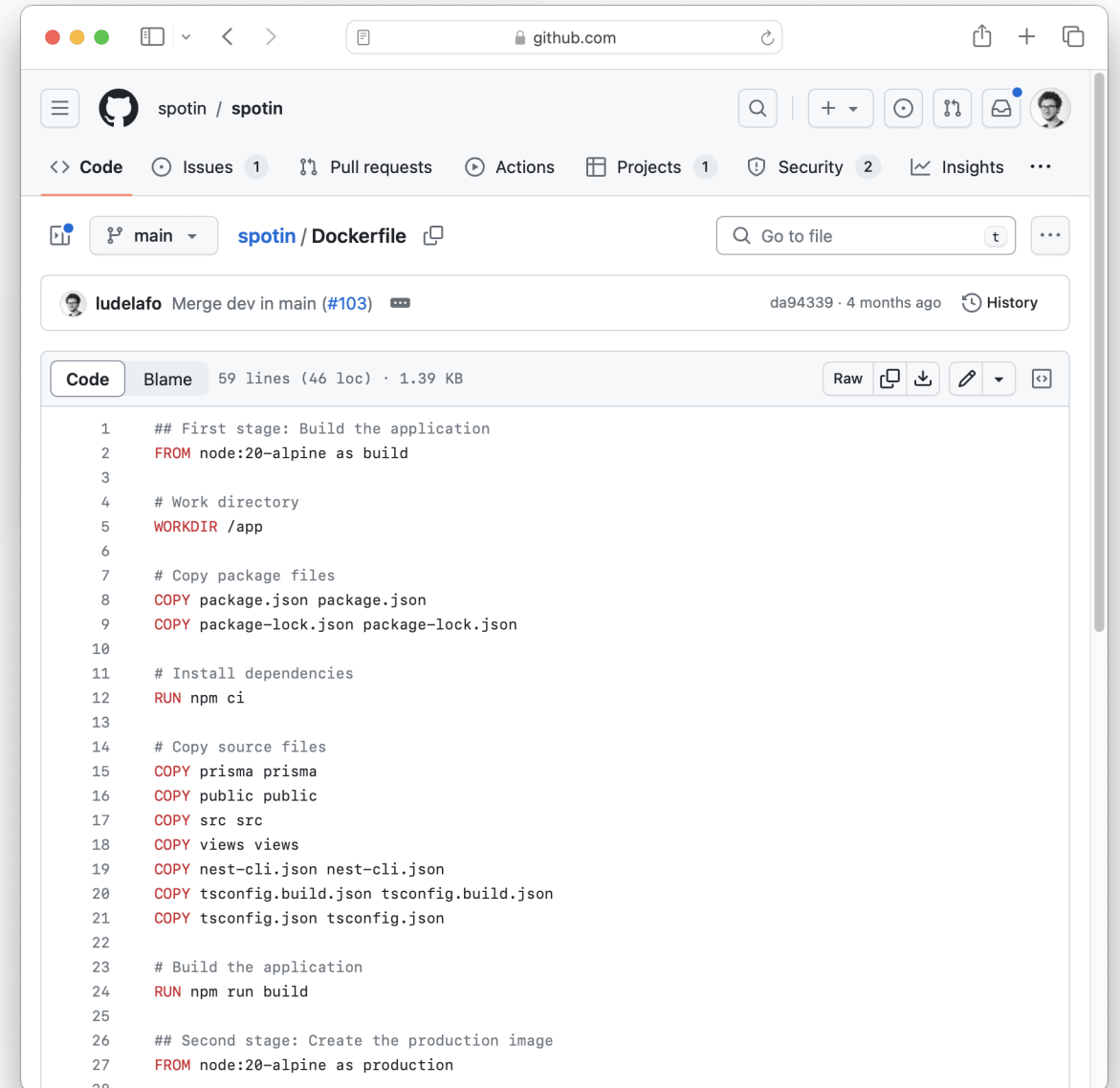
Docker

- Created in 2013
- Container engine
- Composed of two parts:
 - Docker daemon (background process)
 - Docker CLI
- Can be used to build, run and publish containers



Dockerfile specification

- Build a Docker image
- Based on an existing image
- Defines a set of instructions to build the image
- Written in plain text



```
1  ## First stage: Build the application
2  FROM node:20-alpine as build
3
4  # Work directory
5  WORKDIR /app
6
7  # Copy package files
8  COPY package.json package.json
9  COPY package-lock.json package-lock.json
10
11 # Install dependencies
12 RUN npm ci
13
14 # Copy source files
15 COPY prisma prisma
16 COPY public public
17 COPY src src
18 COPY views views
19 COPY nest-cli.json nest-cli.json
20 COPY tsconfig.build.json tsconfig.build.json
21 COPY tsconfig.json tsconfig.json
22
23 # Build the application
24 RUN npm run build
25
26 ## Second stage: Create the production image
27 FROM node:20-alpine as production
28
```

Code examples

Check the code examples in the `heig-vd-dai-course-code-examples` Git repository:

- Basic Dockerfile
- Dockerfile with command
- Dockerfile with entrypoint and command
- Dockerfile with run and copy commands
- Dockerfile with build arguments

Summary

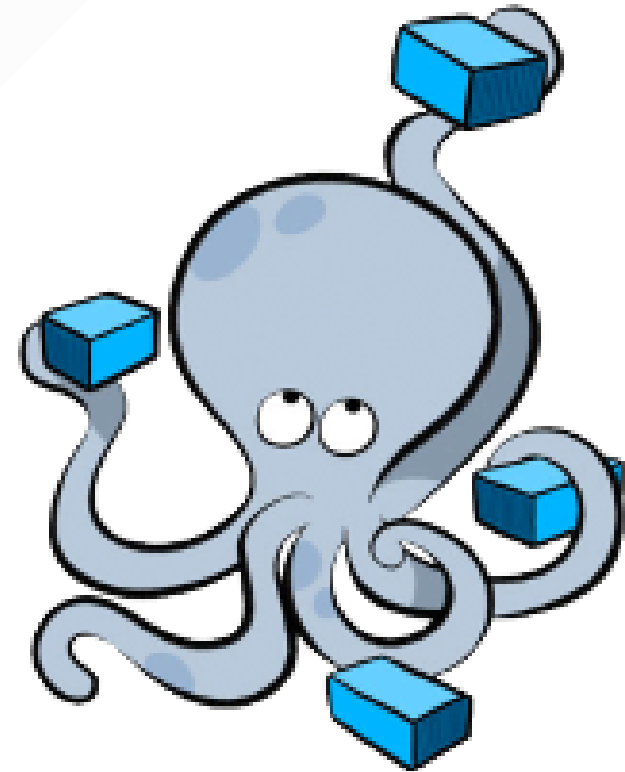
- Docker is a container engine composed of two parts: the Docker daemon and the Docker CLI
- The Docker CLI is used to manage containers and images
- The Dockerfile specification defines a standard for building Docker images
- A Dockerfile is used to build a Docker image
- A Docker image is used to create a container
- A container is a runnable, isolated, instance of an image

Docker Compose

More details for this section in the [course material](#). You can find other resources and alternatives as well.

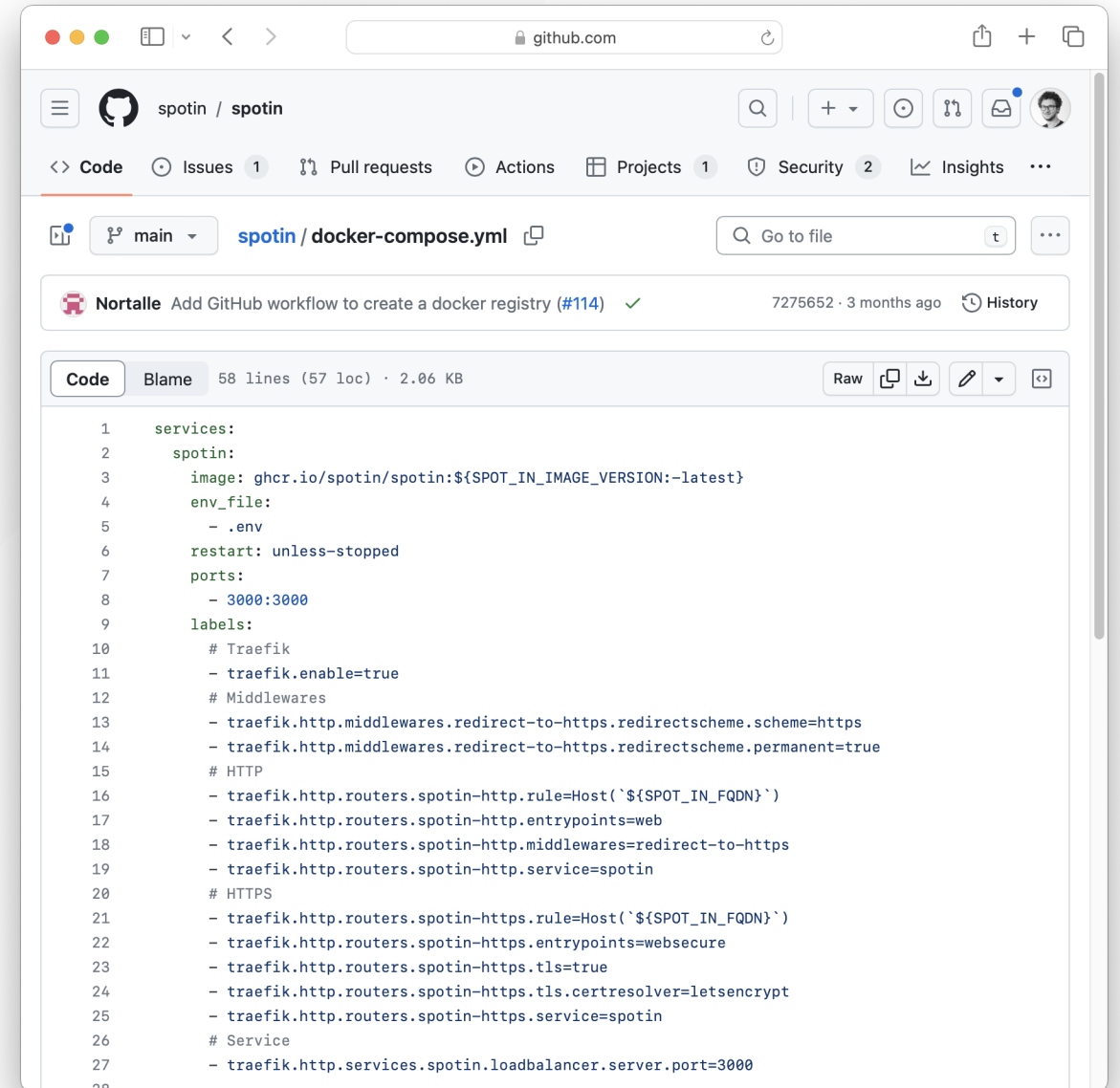
Docker Compose

- Can be used to deploy a multi-container application
- Can be committed with the application
- Can be used to deploy the application on any Docker host
- Easy to use



Docker Compose specification

- Defines the application
 - Services: containers
 - Volumes: shared directories
 - Networks: network communication
- Written in YAML



The screenshot shows a GitHub repository page for a file named `docker-compose.yml`. The file content is as follows:

```
1  services:
2    spotin:
3      image: ghcr.io/spotin/spotin:${SPOT_IN_IMAGE_VERSION:-latest}
4      env_file:
5        - .env
6      restart: unless-stopped
7      ports:
8        - 3000:3000
9      labels:
10       # Traefik
11       - traefik.enable=true
12       # Middlewares
13       - traefik.http.middlewares.redirect-to-https.redirectscheme.scheme=https
14       - traefik.http.middlewares.redirect-to-https.redirectscheme.permanent=true
15       # HTTP
16       - traefik.http.routers.spotin-http.rule=Host(`${SPOT_IN_FQDN}`)
17       - traefik.http.routers.spotin-http.entrypoints=web
18       - traefik.http.routers.spotin-http.middlewares=redirect-to-https
19       - traefik.http.routers.spotin-http.service=spotin
20       # HTTPS
21       - traefik.http.routers.spotin-https.rule=Host(`${SPOT_IN_FQDN}`)
22       - traefik.http.routers.spotin-https.entrypoints=websecure
23       - traefik.http.routers.spotin-https.tls=true
24       - traefik.http.routers.spotin-https.tls.certresolver=letsencrypt
25       - traefik.http.routers.spotin-https.service=spotin
26       # Service
27       - traefik.http.services.spotin.loadbalancer.server.port=3000
28
```

Code examples

Check the code examples in the `heig-vd-dai-course-code-examples` Git repository:

- Basic Docker Compose
- Docker Compose with ports
- Docker Compose with volumes
- Docker Compose with environment variables

Summary

- Docker Compose allows to define a multi-container Docker application in a Docker Compose file
- A Docker Compose file can consist of a set of services, volumes and networks
- A Docker Compose file (`docker-compose.yaml`) can be easily shared and versioned with the application

Questions

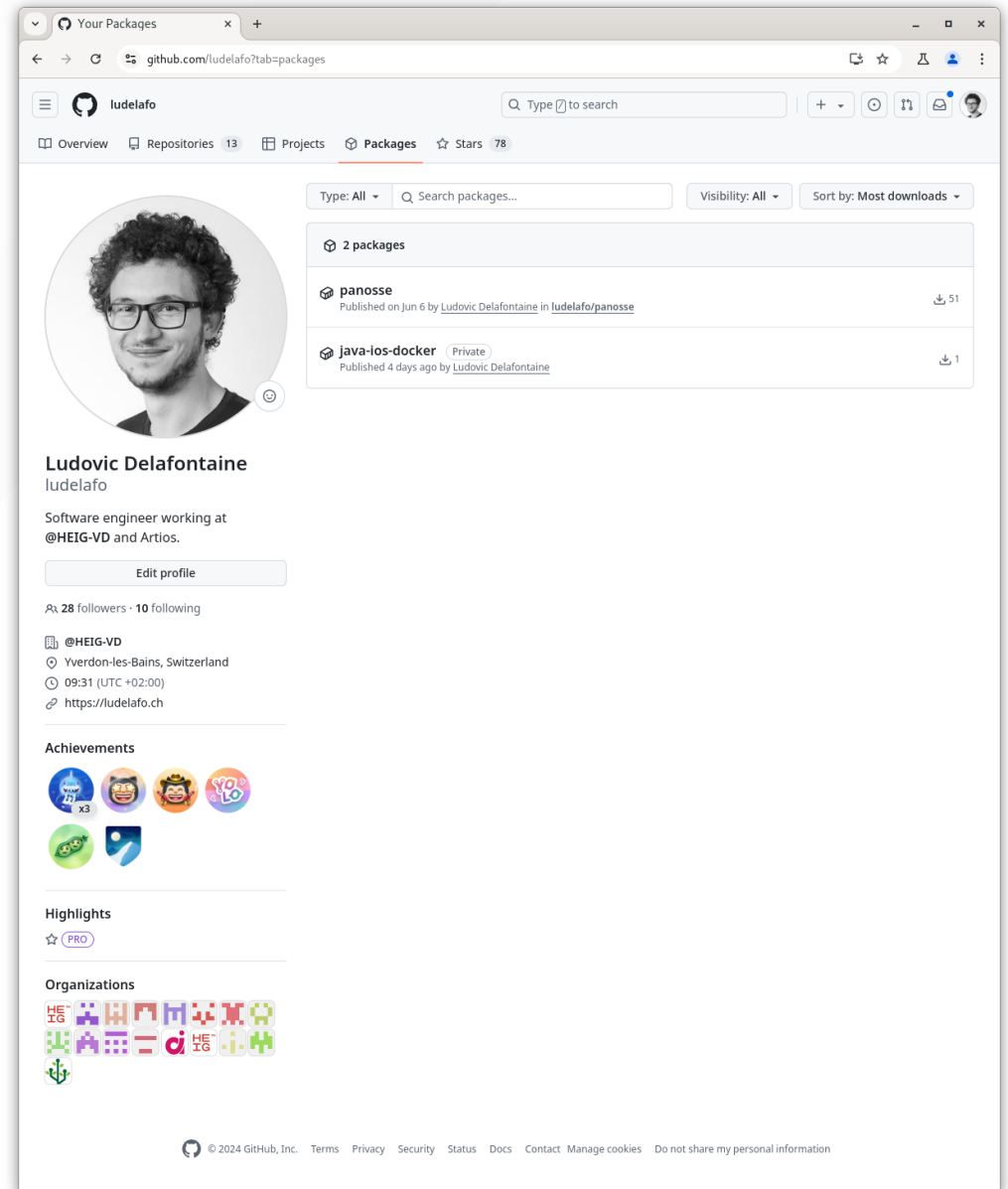
Do you have any questions?

Practical content

What will you do?

Containerize the previous Java IOs project:

- Create the Dockerfile and Docker Compose files
- Publish on GitHub Container Registry
- Run it on any Docker host



Find the practical content

You can find the practical content for this chapter on [GitHub](#).



Finished? Was it easy? Was it hard?

Can you let us know what was easy and what was difficult for you during this chapter?

This will help us to improve the course and adapt the content to your needs. If we notice some difficulties, we will come back to you to help you.

 [GitHub Discussions](#)

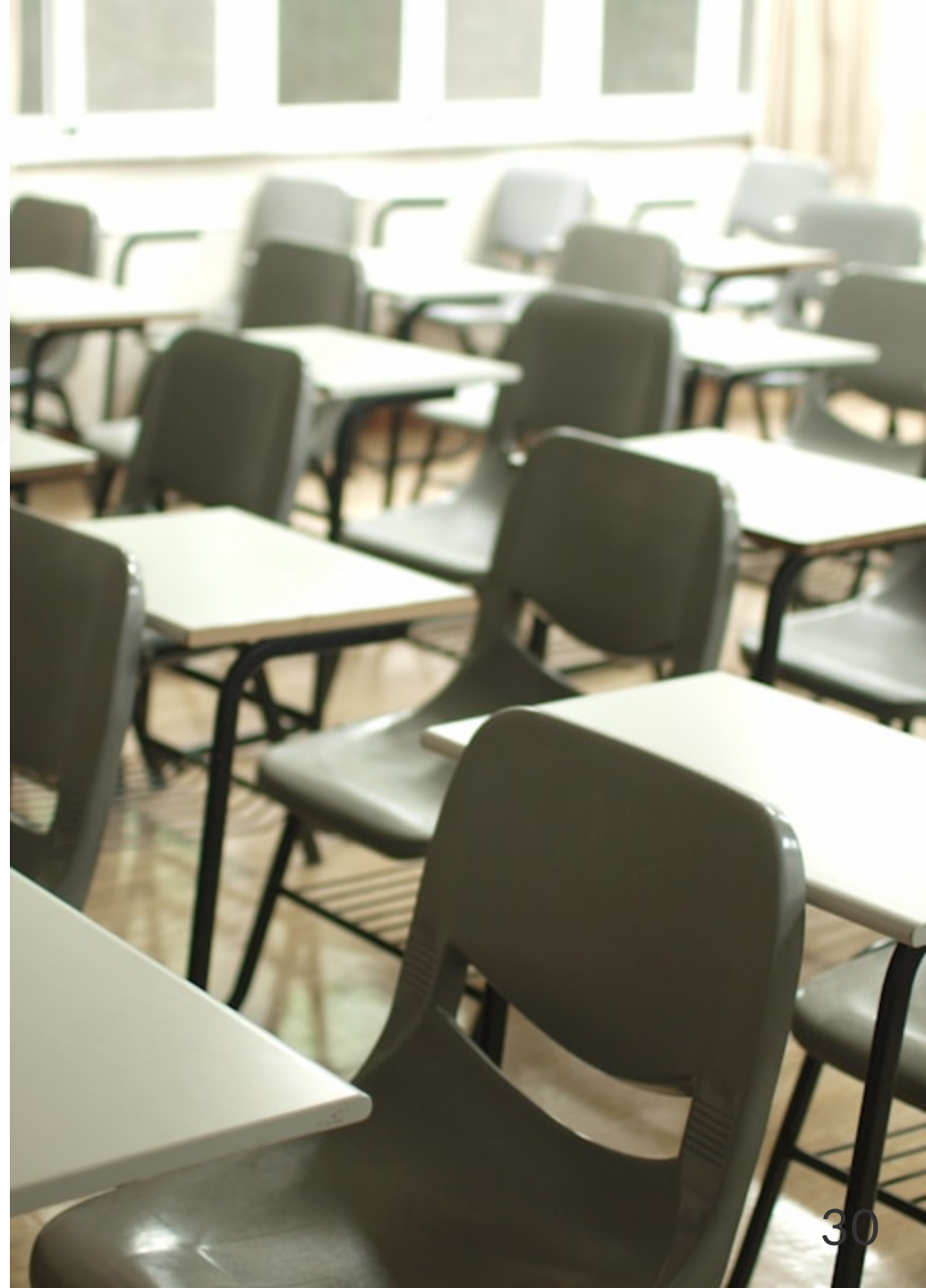
You can use reactions to express your opinion on a comment!

What will you do next?

We are arriving at the end of the first part of the course.

An evaluation will be done to check your understanding of all the content seen in this first part.

More details will be given in the next chapter.



Sources

- Main illustration by [CHUTTERSNAAP](#) on [Unsplash](#)
- Illustration by [Rafif Prawira](#) on [Unsplash](#)
- Illustration by [Taylor Vick](#) on [Unsplash](#)
- Illustration by [Aline de Nadai](#) on [Unsplash](#)
- Illustration by [Scott Webb](#) on [Unsplash](#)
- Illustration by [MChe Lee](#) on [Unsplash](#)