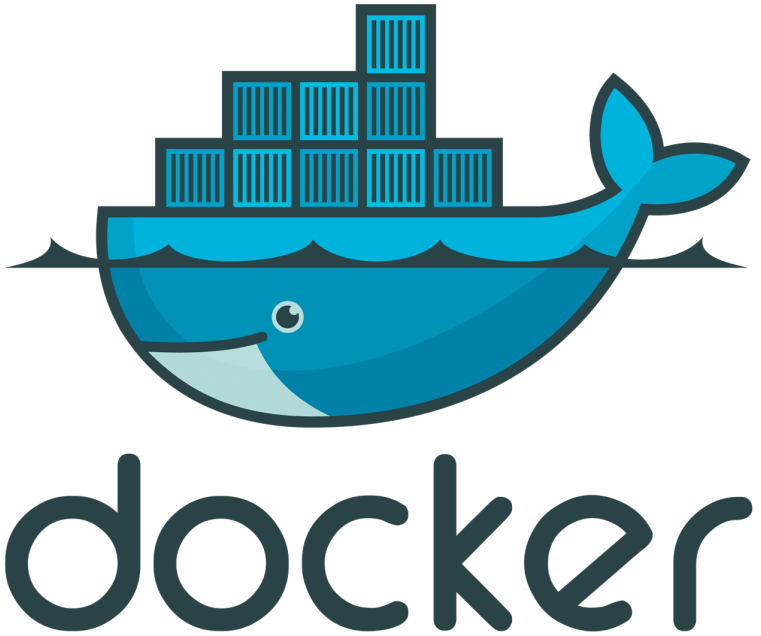
Workshop structure

1. Images
2. Containers
3. Basic syntax
4. Environmental variables
5. Port mapping
6. CI/CD pipeline
7. CI/CD templates

Motivation

- *What is Docker? Why should I learn it?*
- Docker bundles your application and its dependencies into a portable image
- This image can be executed in any environment by running in a container



Dockerization

"Dockerized" (containerized) application:

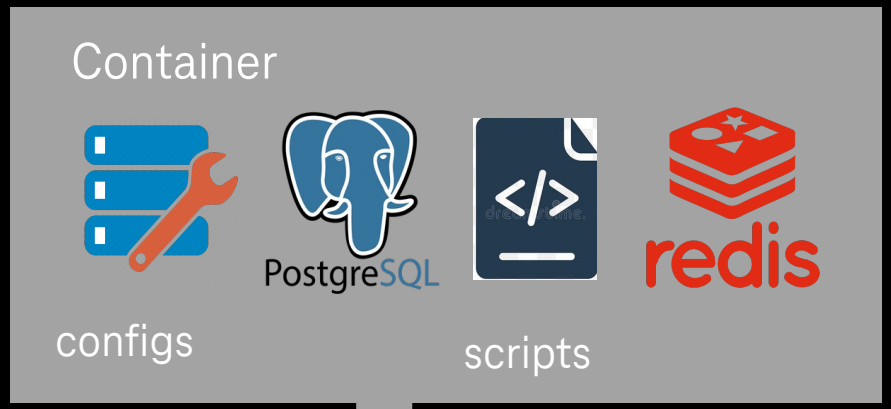
- is **portable**, it should run everywhere the same way
- has all **dependencies** (pip packages, libraries) bundled together with the code
- is **isolated** from the host environment
- *should* run the same way on Windows, Mac and Linux
- is separated into **layers**

Goal of this workshop

- Python files/notebooks → deployable code
- Understand *why* we're using **Docker**
- *How* to bundle your code with all its dependencies
- *How* to version this bundle of code in a registry
- *How* to automatically **test it** and **deploy it**
- *How* to run multiple applications that are interacting with each other in a cool way™

Development

With Docker



run single command;
very close to
production environment


Any OS


Dev *

Without Docker

Installation involves many
different steps


Mac


dev 1

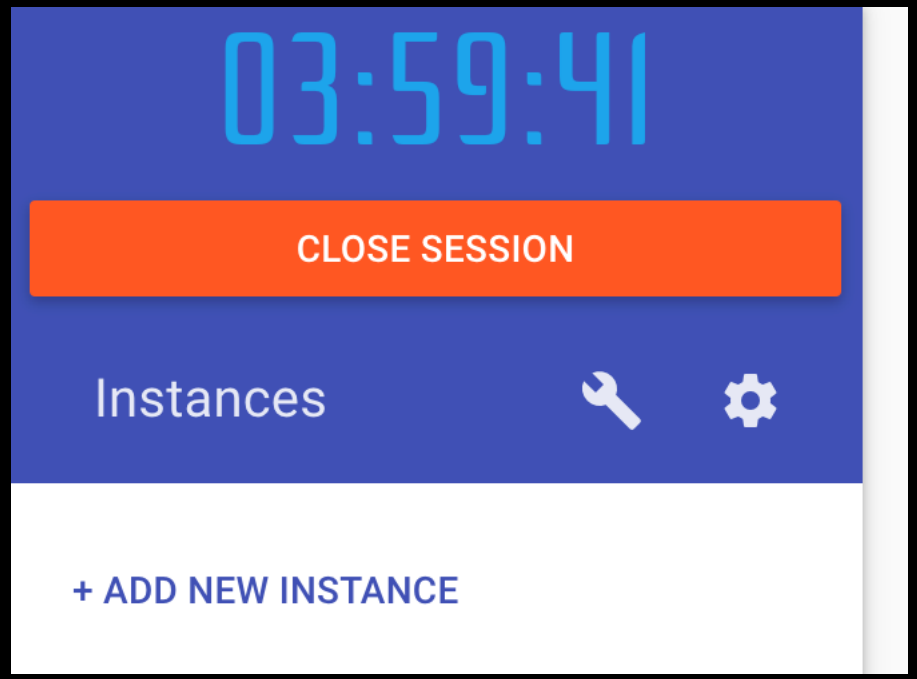

Mac


dev 2

Play With Docker Lab

https://labs.play-with-docker.com

- Creating Instances:



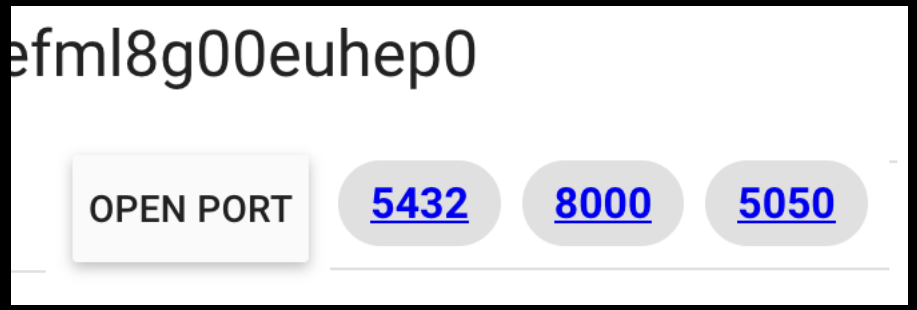
- Working with terminal:

```
[node1] (local) root@192.168.0.28 ~
$ cd code
[node1] (local) root@192.168.0.28 ~/code
$ git clone https://github.com/kuhi/docker-workshop.git
Cloning into 'docker-workshop'...
remote: Enumerating objects: 43, done.
remote: Counting objects: 100% (43/43), done.
remote: Compressing objects: 100% (34/34), done.
remote: Total 43 (delta 10), reused 33 (delta 4), pack-reused 0
Receiving objects: 100% (43/43), 8.18 KiB | 598.00 KiB/s, done.
Resolving deltas: 100% (10/10), done.
[node1] (local) root@192.168.0.28 ~/code
$
```

Play With Docker Lab

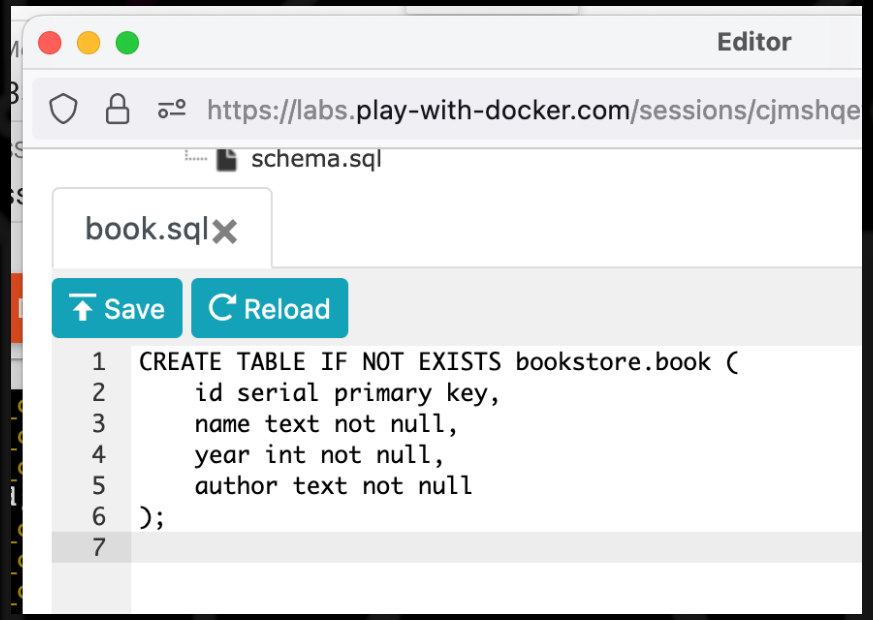
- Ports are opened automatically

```
Application startup complete.
Uvicorn running on http://0.0.0.0:8000
```



- Built-in text editor(s):

```
1 Hello from vim, to start typing, press "i" (insert)
2 To exit without changes, press Esc and type :q!
3 To exit with accepting changes, press Esc and type :x
```



Let’s explore the lab together

Hello World: Exercise 0

1. **Pull the hello-world image from docker hub**
   - `docker pull hello-world`
2. **Run the hello-world image**
   - `docker run hello-world`
3. **According to the output from the hello-world run, try to run an ubuntu image.**
4. **Display a list of images.**
   - `docker ps -a`

1. Images

Docker Image

- An *image* is a collection (or bundle) of things your code requires to execute and your code
- You can **build** the image (building = downloading everything into one folder)
  - **`docker build -t docker-workshop .`**
    - Builds an image called **docker-workshop**
    - Looks in the current directory for a Dockerfile (don't forget the dot at the end!)
- Run **`docker images`** to list all images and tags present on your machine

Build a Docker image

cat Dockerfile

```
FROM ubuntu
CMD ["echo", "Hello World!"]
```

docker build -t docker-workshop .

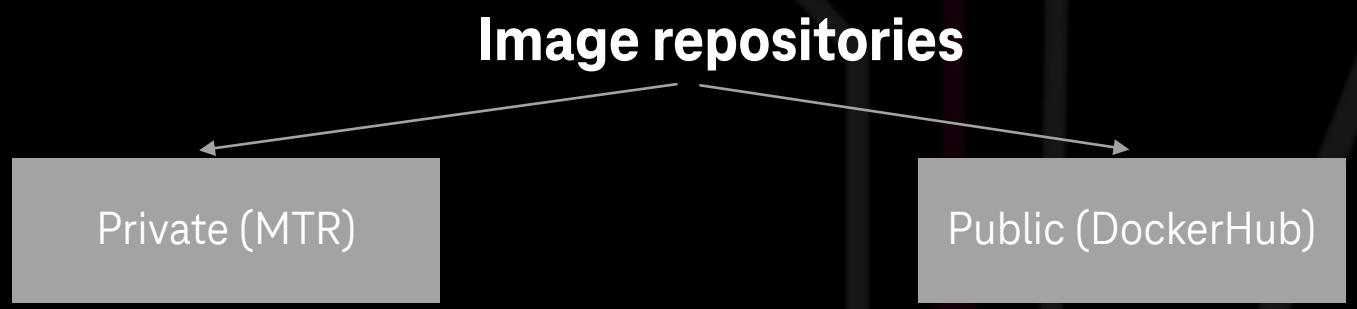
```
[+] Building 0.9s (6/6) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 36B
…
```

docker images

```
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
docker-workshop     latest    5a140a695db4   2 minutes ago    70.1MB
…
```

Image registry

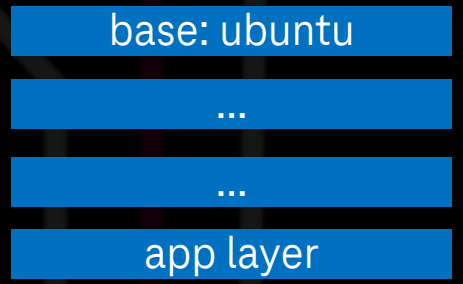
- Images can be stored on a server in the repository – similar as source code in Git
- You can push the image into registry (`docker push`)
- You can pull the image from registry to have it available locally (`docker pull`)
- Each image can have different tags
  - `docker build -t image-name .  # create an image with default tag latest`
  - `docker build -t image-name:1.0.0 . # create an image with tag 1.0.0`



2. Containers

Docker Container

- Container is an isolated runtime environment
- An image can be run in one or more containers (always one application instance per container)
- The container provides a specific ways to communicate with the outside environment:
  - Environmental variables
  - Port mapping
  - Volumes



Run a Docker container

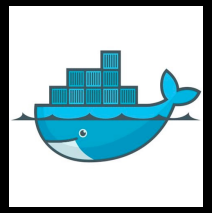
```
docker run docker-workshop          # random name is assigned --name is not used
Hello World!
```

```
docker ps -a
CONTAINER
ID  IMAGE             COMMAND                  CREATED         STATUS                     daf67
4df6d7a  docker-workshop  "echo 'Hello World!'"   2 seconds ago  Exited (0) 1 sec ago  …
```

```
docker run --name greeter docker-workshop
Hello World!
```

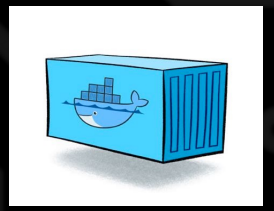
```
docker ps -a
CONTAINER ID  IMAGE             COMMAND                  ...  NAMES
6223c0b77c2a  docker-workshop  "echo 'Hello World!'"  ...  greeter
daf674df6d7a  docker-workshop  "echo 'Hello World!’”  ...  nifty_lalande
```

Image vs. Container



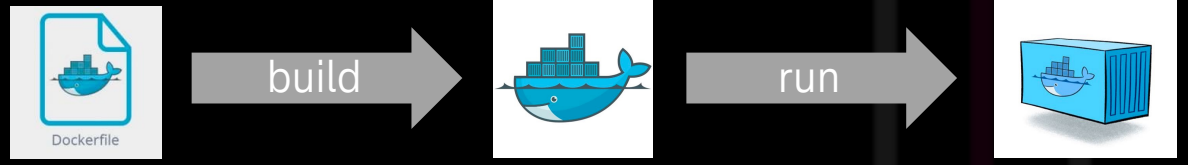
Image

the actual package, the actual immutable artifact



Container

virtualized run-time environment for the image



Analogy

- Docker vs Git analogy

Docker	Git
Image	Source code
Container	Running application
Tag	Branch
Registry (DockerHub, MTR)	GitHub, GitLab
Environmental variables	Program arguments

- Objected-oriented programming analogy
  - image = class
  - container = instance

3. Basic syntax

Basic Syntax: Dockerfile

- FROM – instruction initializes a new build stage and sets the base image for subsequent instructions
- CMD – instruction executes the command (provides defaults cmds for executing container)
- RUN – instruction will execute a commands
- ENV – instruction sets the environment variable <key> to the value <value>
- COPY – instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>
- ENTRYPOINT – allows you to configure a container that will run as an executable
- VOLUME – instruction creates a mount point with the specified name

Basic Syntax: CLI

build – Build an image from a **Dockerfile**

run – create and run a new container from image

ps – list containers

push/pull – upload/download image to/from registry

Example Dockerfile

```
FROM python:3.11

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["python", "-m", "app.main"]
```

use an official Python docker image

copy the requirements file into the image

install the pip dependencies

copy all files to the image

declare that app runs on port 8000

run the application

Exercise 1: Dockerize FastAPI app

1. Open the repository https://github.com/kuhi/docker-workshop
   - `git clone https://github.com/kuhi/docker-workshop`
   - `cd docker-workshop`
2. Switch to the branch exercise1
   - `git checkout exercise1`
3. Follow the instructions in Readme

You can find the solution in the `exercise1-solution` branch

4. Environmental variables

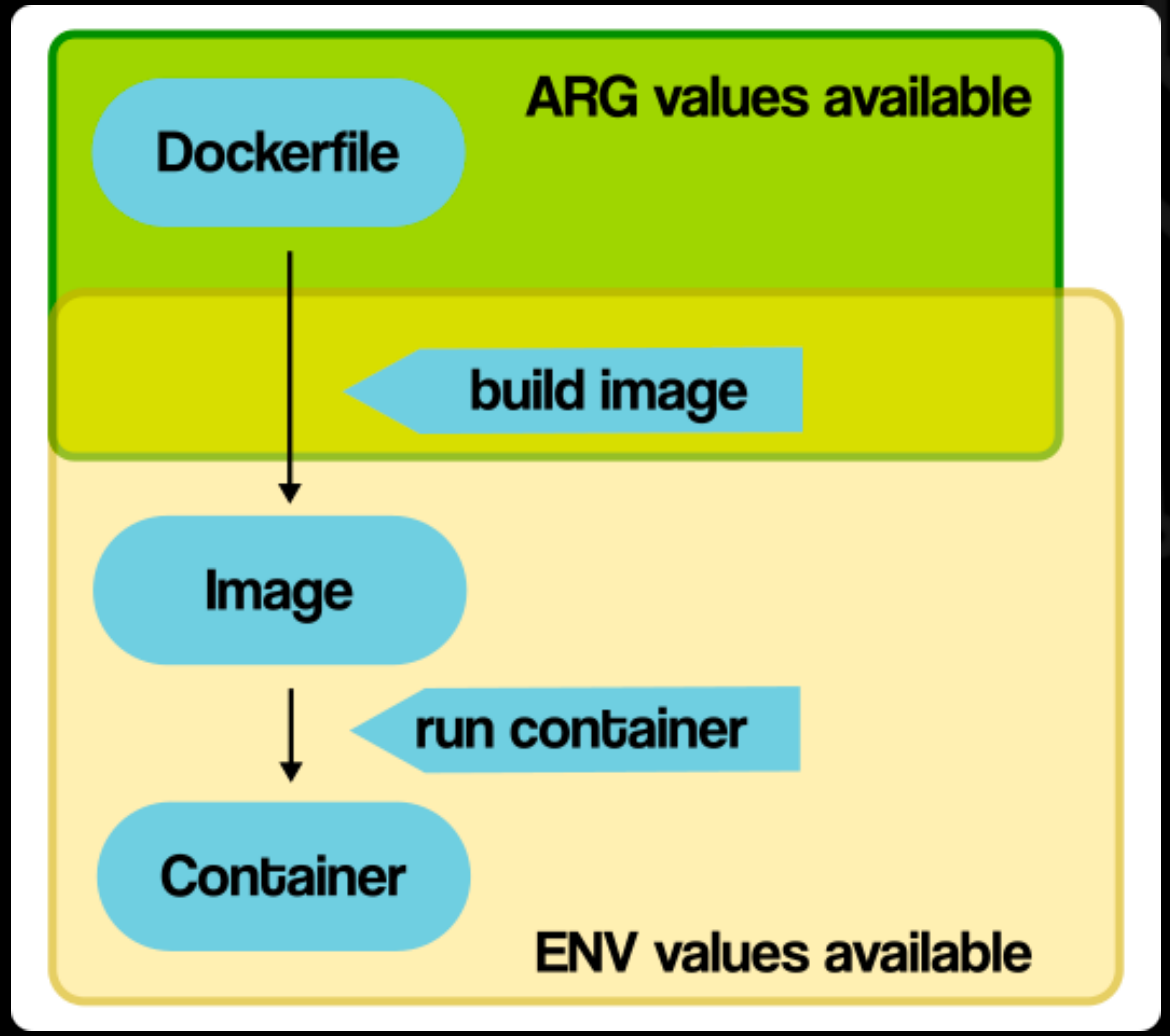
What are environmental variables?

Environmental variables are dynamic values that can be set and accessed within a system or application to configure behaviour or store important information.

Environmental variables in Docker:

- Dynamic key-value pairs used for configuring containerized applications.
- **Flexibility**: Decouple application configuration from the container.
- **Injection**: Set during runtime using Dockerfile or `docker run -e <key>=<value>`
- **Accessibility**: Accessed programmatically within the application.
- **Security**: Safely store sensitive information without exposing it in the container image.

Environmental variables in Docker



- **ENV:**
  - Available during runtime
  - Used to define standard environmental variables
  - Can be changed using the *docker run –e* flag
- **.env:**
  - File with key value pairs (KEY=value\nKEY2=value2)
  - Each key becomes an environmental variable
  - Docker Compose-specific thing (in the context of Docker)

Environmental Variables: Mini-Exercise

1. Copy and paste the following code into a python file (e.g. hello.py):

```
import os

NAME = os.getenv("NAME", "World")
print(f"Hello, {NAME}")
```

2. Run this script in your terminal, try altering the environmental variable (NAME):

```
➢ python hello.py

➢ NAME=Joker python hello.py                # on MacOS (Unix)

➢ $env:NAME="Batman"; python hello.py       # on Windows
```

5. Port mapping

Port mapping in Docker

By default, no ports inside the container are available to the host machine (complete isolation)

Port mapping in Docker enables containers to communicate with the external world by exposing and forwarding network ports.

Port mapping allows binding container ports to specific ports on the host machine.

- **Connectivity**: By mapping container ports to host ports, external systems can communicate with containerized applications.
- **Flexibility**: Port mapping enables running multiple containers on the same host, each using the same port number but different host port bindings.

Port mapping in Docker

- Port mapping is specified using the `-p <host-port>:<container-port>` (or `--publish ...`) flag in the *docker run* command or Docker Compose config.
- **Example**: To map container port 80 to host port 8000, the syntax would be *-p 8080:80*.

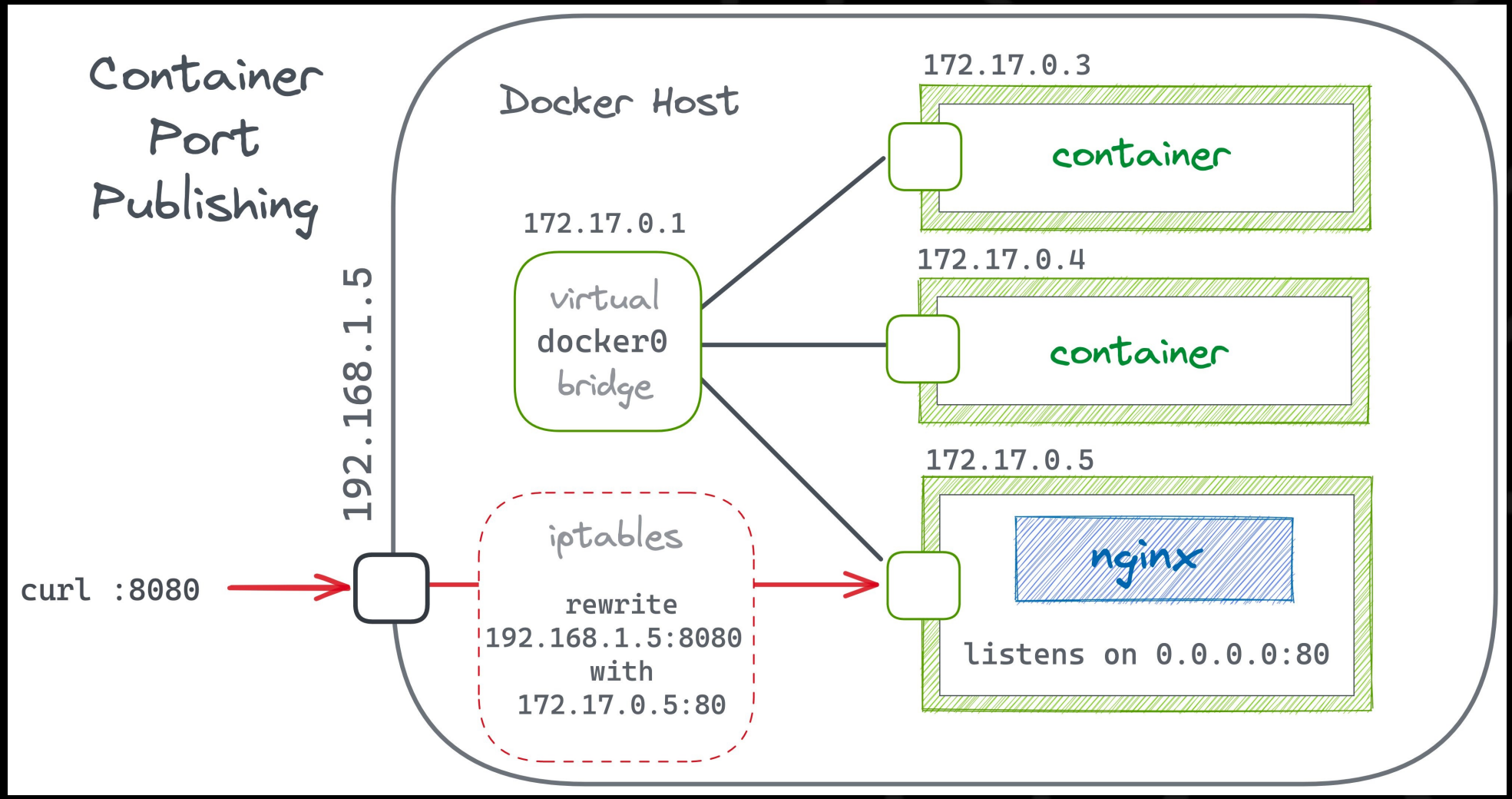
```
docker run --publish 8080:80 --rm nginx
```

See the application running in your browser: **http://127.0.0.1:8080/**

**nginx is a HTML server running on port 80*

A DIAGRAM!

- @iximiuz on Twitter



Exercise 2: Application with database

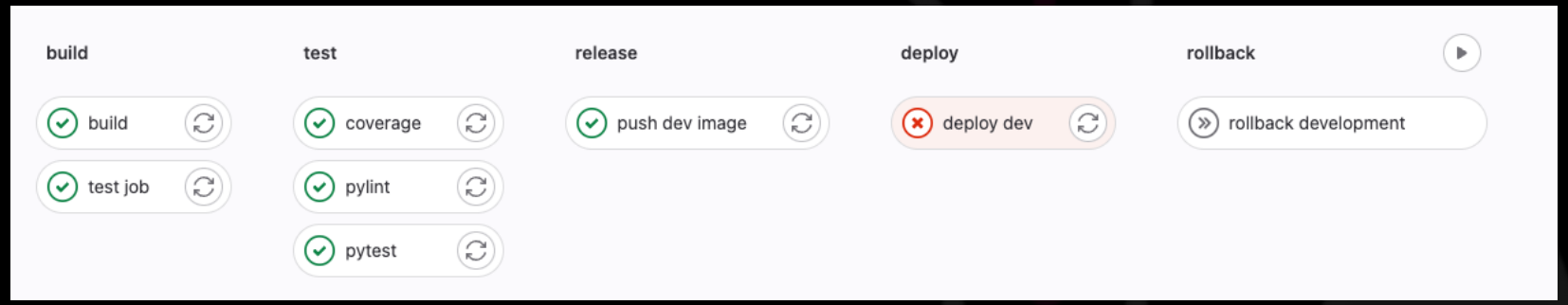
1. Open the repository **https://github.com/kuhi/docker-workshop**
2. Switch to the branch `exercise2`
   - `git checkout exercise2`
3. Follow the instructions in Readme

You can find the solution in the `exercise2-solution` branch

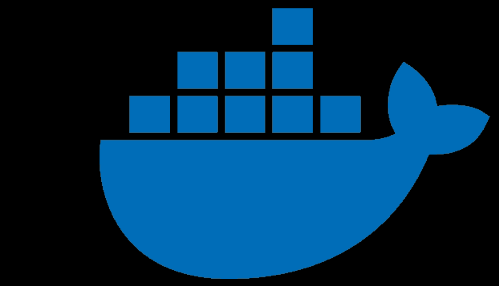
6. CI/CD pipeline

CI/CD pipeline

- **CI/CD (Continuous Integration/Continuous Delivery) is there for you to automate everything after you commit the code**
- Pipelines comprise of:
  - Jobs, which define *what* to do. For example, jobs that compile or test code.
  - Stages, which define *when* to run the jobs. For example, stages that run tests after stages that compile the code.
- Each stage can have multiple jobs, which run in parallel



Project tooling



Docker encapsulates the app in an isolated container, which can be run anywhere. The app should be *transportable* and its environment *predictable*.

Pytest runs your unit tests automatically. You just need to place a file called test_*.py with test_* functions in your project.

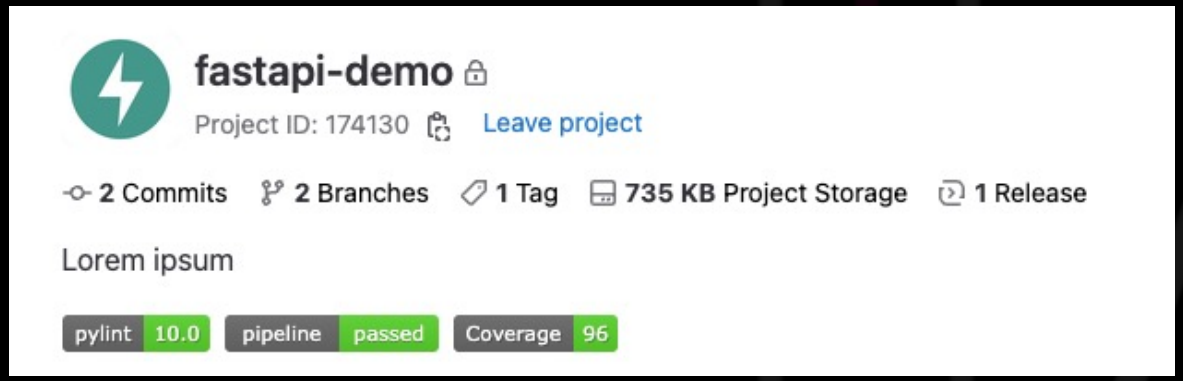
Coverage measures your code test coverage. It can give you hint which lines are not yet tested.

Pylint checks your code for common bad habits and rates the code on a scale from 0 to 10.

7. CI/CD templates

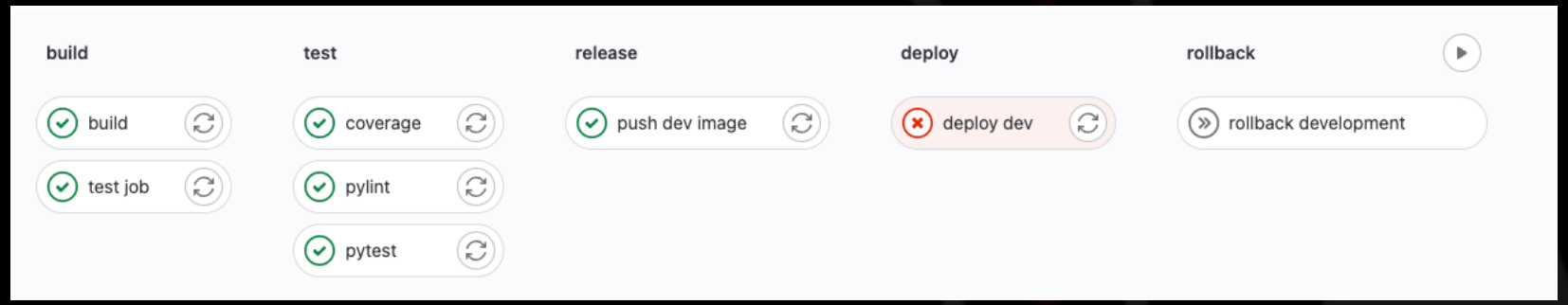
Repository templates

- **You can use GitLab repository templates to generate a new project with the full tooling setup**
- https://gitlab.devops.telekom.de/dtse-cz/repo-templates
- Currently there are two templates:
  - Basic Python template
  - FastAPI template for REST APIs in Python
- More templates are to come (tell us what you‘d like!)



Repository template features

- CI/CD pipeline with ***build***, ***test***, ***release*** and ***deployment*** stages set up
- Automatic image versioning and pushing to MTR
- Automatic deployments to development server from the *develop* branch
- Manually triggerable deployments to production server from the *main* branch
- Manually triggerable *rollback* job in case something breaks



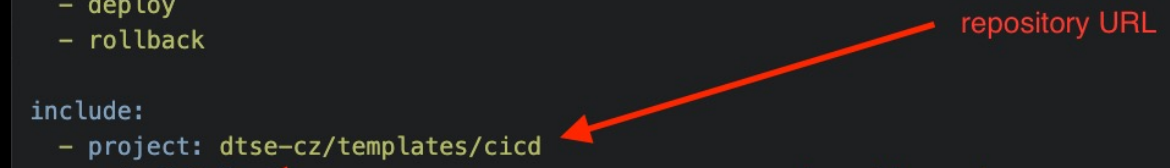
Templates (CI/CD)

There is a repository with reusable CI/CD pipeline jobs for you

- https://gitlab.devops.telekom.de/dtse-cz/templates/cicd
- The usage should be clear from the FastAPI template
- Current version 1.5.0 is tested on many projects
- Full documentation in Readme.md
- Sample config files in the `examples/` folder

Templates (CI/CD)

```yaml
stages:
  - build
  - test
  - release
  - deploy
  - rollback

include:
  - project: dtse-cz/templates/cicd
    ref: 0.5.0
    file: jobs.yml

# build job
build:
  extends: .build

# push the development image
push dev image:
  extends: .push-dev-image

# push the production image
push image:
  extends: .push-image

# testing
pylint:
  extends: .pylint
```



- You need to include the CI/CD template from repository
- Always pin to a version to be safe from breaking changes!
- Extend the template jobs to use them
- Configuration is done using the variables block

Exercise 3: CI/CD templates

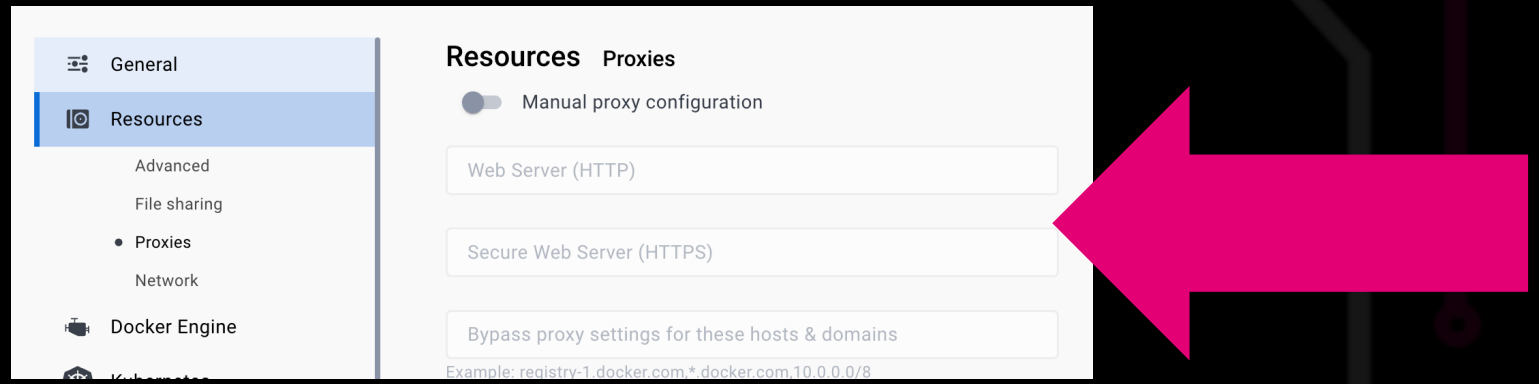
1. Create a new repository in GitLab
    1. Go to the **Docker Workshop** group (gitlab.devops.telekom.de/dtse-cz/docker-workshop)
    2. **New Project** -> **Create from template** -> **Group**
    3. Choose the **FastAPI template** and click the **Use template** button
    4. Use your name in the name of repository
2. Create a new branch from `develop`
3. Create a MR merging this branch back to the `develop` one
4. Follow the Readme in this repository to create a distinct application name (*workshop-yourname*)
5. Make some small changes to the code
6. Go to Build → Pipelines → Run pipeline, select your branch name and run it
7. Wait for the pipeline to finish and go to this URL to see your application:
    - **https://*workshop-yourname*.apps.ocp501.tc-npr.aws.telekom.de/**

Courses on Docker

- Bret Fisher does the best courses on Docker
- A ton of free material is available on YouTube
  - https://www.youtube.com/@BretFisher
- If you're interested in Docker, go see his basic course
  - https://www.bretfisher.com/courses/ - courses, discounts
- Udemy has discounts all the time

Bonus: DTSE Windows specifics

- *”Life would be too easy without a shitty company proxy”* – author unknown
- When on Windows, run commands in Powershell:
  - For pip install prepend your command with `$env:HTTP_PROXY="http://10.255.46.226:3128"; $env:HTTPS_PROXY="http://10.255.46.226:3128"; $env:NO_PROXY="t-internal.com";`
  - For docker build, add `--build-arg http_proxy=http://10.255.46.226:3128 --build-arg https_proxy=http://10.255.46.226:3128 --build-arg no_proxy="t-internal.com"`
  - More info on proxies https://wiki.telekom.de/pages/viewpage.action?pageId=1340409169
- In Docker Desktop, set your proxies in the settings



Q&A discussion

https://forms.gle/ADHvbUeAFNbdHQnW7

We’d love your feedback

❤️