Dockerfile and building images

Basic concepts

Corso Docker e Orchestrazione di Container
Seconda edizione

Diego Ciangottini INFN-PERUGIA

Why do we need dockers

A short list of benefits includes:

- **Faster development process**
  a. There is no need to install 3rd-party apps (i.e. DBs) on the system -- you can run it in containers.
  b.   Keep multiple versions of the same app on one host OS
- **Handy application encapsulation**
  a. Docker gives you a unified image format to distribute you applications across different host systems and cloud services. You can deliver your application in one piece with all the required dependencies (included in an image) ready to run.
- **Same behaviour on local machine / dev / staging / production servers**
  a. reduces to almost zero the probability of error caused by different versions of operating systems, system-dependencies, etc. Your application will use the same base image with the same OS version and the required dependencies.
- **Easy and clear monitoring**
  a. Out of the box, you have a unified way to read log files from all running containers. You don't need to remember all the specific paths where your app and its dependencies store log files and write custom hooks to handle this.
- **Easy to scale**

Today in this session we will focus on building images

1. Docker containers are runtime environments. You usually run one main process in one Docker container. You can think of this like one Docker container provides one service in your project.
   a. For example you can start one container to be your MySQL database and start another container to be your Wordpress server and connect these containers together to get a Wordpress project setup.
   b. You can start containers to run all the tech you can think of, you can run databases, web servers, web frameworks, test servers, execute big data scripts, work on shell scripts, etc.

2. Docker containers are started by running a Docker image. A Docker image is a **pre-built environment for a certain technology or service.** A Docker image is not a runtime, it's rather a collection of files, libraries and configuration files that build up an environment

Recap Image VS Container

- **Image**: Is a blueprint for what you want to build. Ex: Ubuntu + TensorFlow with Nvidia Drivers and a running Jupyter Server.

- **Container**: Is an instantiation of an image that you have brought to life. **You can have multiple copies of the same image running**. It is really important to grasp the difference between an image and a container as this is a common source of confusion for newcomers.

Our main objective is to build images

Or better we want to build our image, to run our application/service. Two paths:

- Building images interactively
    - Create a container from a base image.
    - Install software manually in the container, and turn it into a new image.
    - Learn about new commands: docker commit, docker tag, and docker diff.
- Building docker images with Dockerfiles
    - Write a Dockerfile.
    - Build an image from a Dockerfile.
    - Containerize a Python App

Workplan

1. Create a container (with docker run) using our base distro of choice.
2. Run a bunch of commands to install and set up our software in the container.
    a. review changes docker diff, turn the container into a new image with docker commit, tag the image docker tag.
3. All nice but how to do it "properly"? : The Dockerfile
    a. From zero to a simple containerized Python App
4. Share our services (Dockers): The Registry
    a. We'll see Dockerhub

A bit more in details: today we will go through

- **Dockerfile**: Recipe for creating an Image. Dockerfiles contain special Docker syntax. From the official documentation: A Dockerfile is a text document that contains all the **commands** a user could call on the command line to assemble an image.

- **Layer**: modification to an existing image, represented by an instruction in the Dockerfile. Layers are applied in sequence to the base image to create the final image.

- **Commit**: Like git, Docker containers offer version control. You can save the state of your docker container at anytime **as a new image** by committing the changes.

- **DockerHub / Image Registry**: Place where people can post public (or private) docker images to facilitate collaboration and sharing.

What is the Dockerfile?

A Dockerfile is a text file that defines a Docker image. You'll use a Dockerfile to create your own custom Docker image, in other words **to define your custom environment** to be used in a Docker container.

- A Dockerfile is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
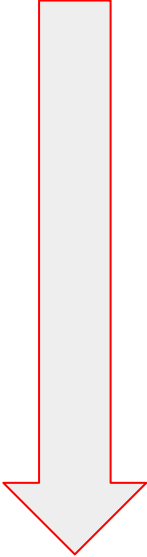- The docker build command builds an image from a Dockerfile

Simple example:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y figlet
```

Why and when you'd want to use a Dockerfile?

You want to create your own Dockerfile **when existing images don't satisfy your project needs.** This will actually **happen most of the time**, which means that learning about the Dockerfile is a pretty essential part of working with Docker.

You'll usually **start searching for available Docker images** on the Docker store [see later] you'll also **find images on github** included with a good number of repos (i**n the form of a Dockerfile**), or you can share Docker images within your team by **creating your own Docker Registry**

Let's do a (small) step ahead

1. The Dockerfile is a text file that (mostly) contains the instructions that you would execute on the command line to create an image.

2. A Dockerfile is a step by step set of instructions.

3. Docker provides a set of standard instructions to be used in the Dockerfile, like FROM, COPY, RUN, ENV, EXPOSE, CMD just to name a few basic ones.

4. Docker will build a Docker image automatically by reading these instructions from the Dockerfile.

Working with dockers

1. You create the Dockerfile and define the steps that build up your images
2. You issue the **docker build** command which will build a Docker image from your Dockerfile
3. Now you can use this image to start containers with the **docker run** command
4. And share…

The main source of Docker images online is the Docker store. You just need to search for your preferred tech component, pull the image from the store with the `docker pull` command and you are ready to start up containers
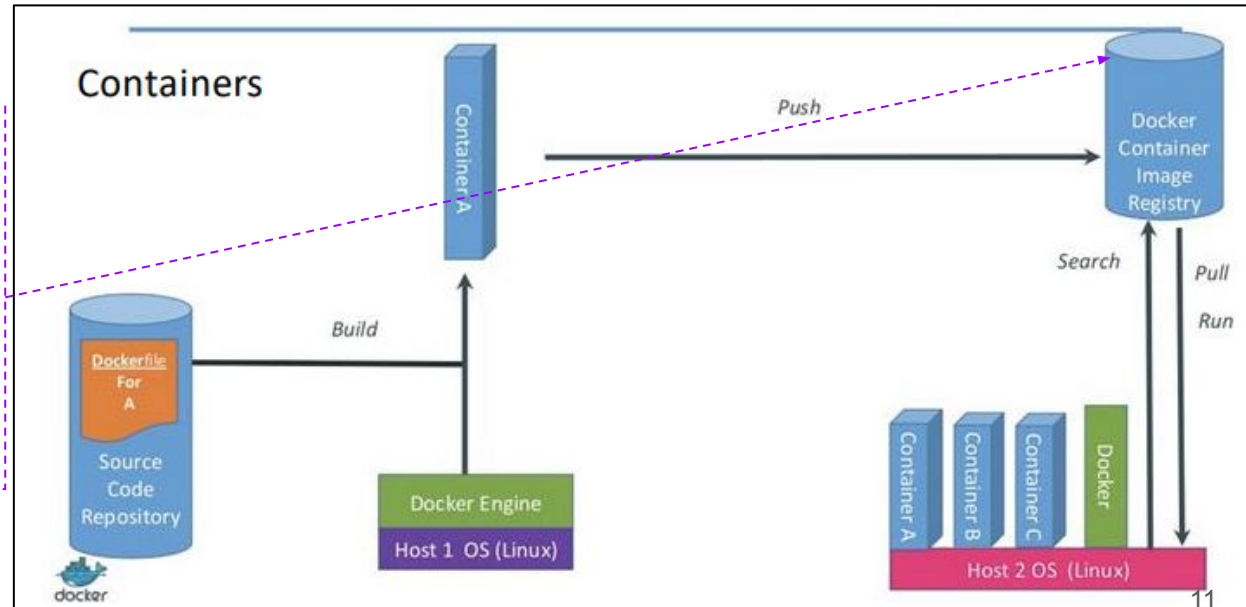
Container registry

A container registry is a docker image repository that allows:
- to distribute docker images
- to have a single place to store images
- integration between image storage and distribution in development workflow

"By default" we rely on DockerHub

Can interact via API for
- Pulling an image: downloads the docker image from the repository
  - docker pull ubuntu
  - docker push ubuntu

Pushing an image: copies the docker image to the repository
- docker push <hub-user> / <repo-name>: <tag>

Public and private registry

Container registry is divided into two types

- **Public:** they are registry for distributing open source images. To publish an image you need to login. (eg DockerHub)
- **Private:** they are generally used for internal developments or releases. For these you need to login for both push and pull images.

For both types it is however possible to define the visibility (public / private) of the individual repositories.

To authenticate to a registry:
- docker login -> DockerHub
- (docker login hostname: port -> private registry)

Tag and image name

Image name consists of
- **hostname [: port]**: hostname and port of the registry, if omitted it means that you are referring to DockerHub
- **name / image**: the image name is an alphanumeric string that can be separated by slash, groups images of the same type
- **tag:** string of up to 128 characters to identify the version of the same type of image

    --> hostname: port / name / image: tag

- A container name and tag can be assigned in two ways
    1. docker tag \<ImageID> [hostname [: port] /] name / image: [tag]
    2. docker build –t [hostname [: port] /] name / image: [tag].

Let's start the Hands-on

https://infn-bari-school.github.io/docker-tutorial/image/intro

Final part of the session

Tips for optimizing Docker builds

Efficient Dockerfiles

❏ Reduce the number of layers.
❏ Leverage the build cache so that builds can be faster.
　　❏ Add context which changes a lot (for example, the source code of your project) at the end of Dockerfile → it will utilize Docker cache effectively.
❏ Include only necessary context → use a .dockerignore file (like .gitignore in git)
❏ Avoid installing unnecessary packages → it will consume extra disk space.
❏ Use environment variables (in RUN, EXPOSE, VOLUME) → It will make your Dockerfile more flexible.
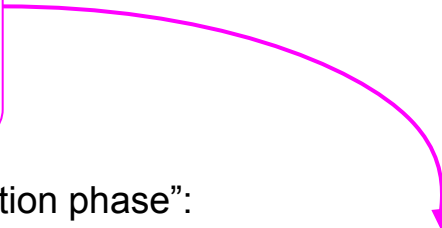
Reducing layers

We learned that:
- Each line in a Dockerfile creates a new layer.
- Build your Dockerfile to take advantage of Docker's caching system.
- Combine commands by using && to continue commands and \ to wrap lines.

Note: it is frequent to build a Dockerfile line by line:

RUN apt-get install this thing
RUN apt-get install that thing and that other one
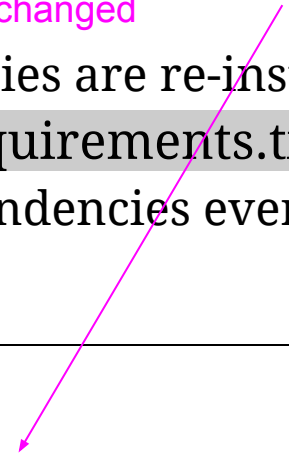**RUN** apt-get install some more stuff

And then refactor it trivially before move to "production phase":

RUN apt-get install this thing and that thing and that other one and some more stuff

Improve caching

- Classic Dockerfile problem:
- "each time I change a line of code, all my dependencies are re-installed!"
- Solution: COPY dependency lists (package.json, requirements.txt, etc.) by themselves to avoid reinstalling unchanged dependencies every time.

```
FROM python
WORKDIR /src
COPY . .
RUN pip install -qr requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

```
FROM python
WORKDIR /src
COPY requirements.txt .
RUN pip install -qr requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

Be careful with chown and move

- operations like chown, mv can be expensive.
- For instance, in the Dockerfile snippet below, each RUN line creates a layer with an entire copy of some-file.

```
COPY some-file .
RUN chown www-data:www-data some-file
RUN chmod 644 some-file
RUN mv some-file /var/www
```

- Instead of using mv, directly put files at the right place (i.e. using tar/zip)
  a.   This will be just 1 layer

Multi-stage images

Multistage builds feature in Dockerfiles enables you to create smaller container images with better caching

It actually answers to a question like this:
- Can I use one (base) image to build my app and another (base) image to run it.

Yes this can be done by using more than one FROM instructions in the Dockerfile.

- At any point in our Dockerfile, we can add a new FROM line.
- This line starts a new stage of our build.
- Each stage can access the files of the previous stages with COPY --from=....
- When a build is tagged (with docker build -t ...), the last stage is tagged.
- Previous stages are not discarded: they will be used for caching, and can be referenced.

Multi-stage builds in practice

- Each stage is numbered, starting at 0
- We can copy a file from a previous stage by indicating its number, e.g.:
- **COPY** --from=0 /file/from/first/stage /location/**in**/current/stage
- We can also name stages, and reference these names:
- **FROM** golang AS builder
- **RUN** ...
- **FROM** alpine
- **COPY** --from=builder /go/bin/mylittlebinary /usr/local/bin/

And now we can try it

If you like just go here: multistage-example

And let's extend the COPY hello.C example that we saw right before.

Ok.. but when I need to optimize an image ?

When authoring official images, it is a good idea to reduce as much as possible:

- the number of layers,
- the size of the final image.

This is often done at the expense of build time and convenience for the image maintainer; but when an image is downloaded millions of time, saving even a few seconds of pull time can be worth it.

But sometimes…

Sometimes, it is better to prioritize *maintainer convenience*.

In particular, if:

- the image changes a lot,
- the image has very few users (e.g. only 1, the maintainer!),
- the image is built and run on the same machine,
- the image is built and run on machines with a very fast link ...

In these cases, just keep things simple!

CI/CD session

CI/CD: the CI thing

- Consider an application that has its code stored in a Git repository in GitLab. Developers push code changes every day, multiple times a day.
- For every push to the repository, you can create a set of scripts to build and test your application automatically. These scripts help decrease the chances that you introduce errors in your application.
- This practice is known as Continuous Integration. Each change submitted to an application, even to development branches, is built and tested automatically and continuously. These tests ensure the changes pass all tests, guidelines, and code compliance standards you established for your application.
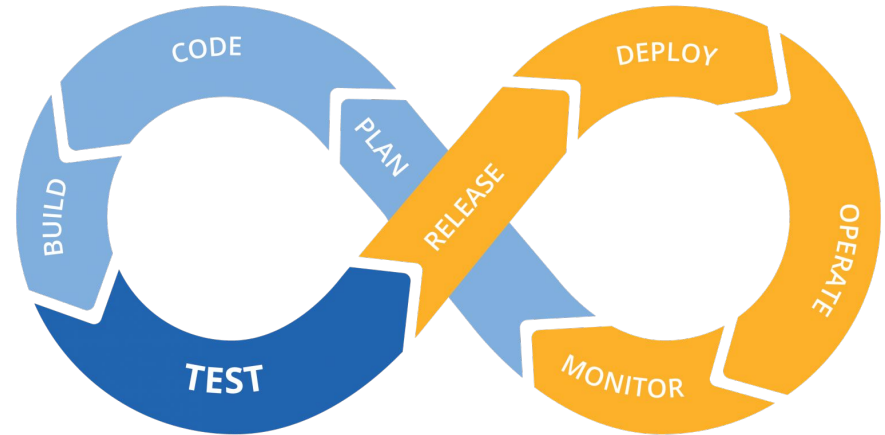
CI/CD: the CD thing

Continuous Delivery is a step beyond Continuous Integration. Not only is your application built and tested each time a code change is pushed to the codebase, the application is also deployed continuously. However, with continuous delivery, you trigger the deployments manually.

Continuous Delivery checks the code automatically, but it requires human intervention to manually and strategically trigger the deployment of the changes.
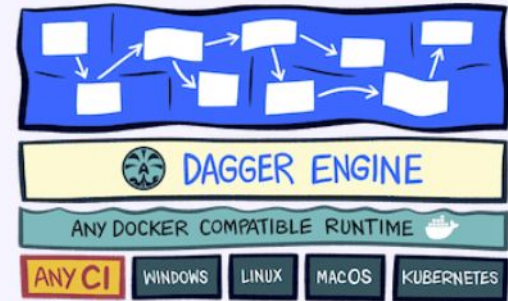
Many tools, similar objectives…
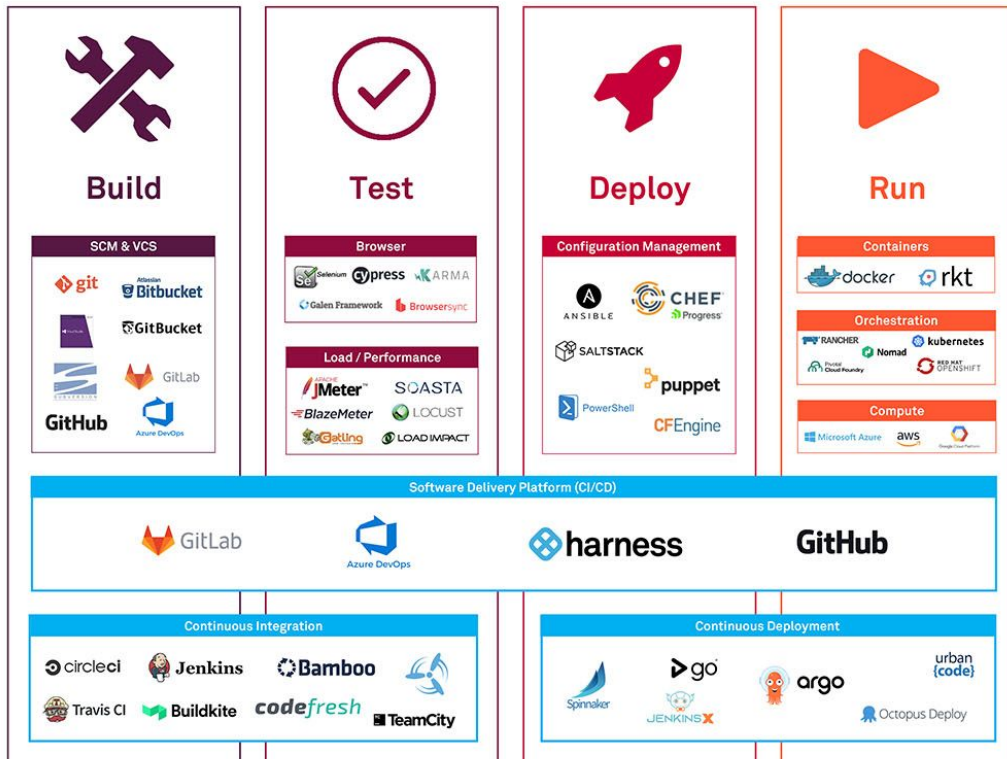
Many tools, similar objectives…
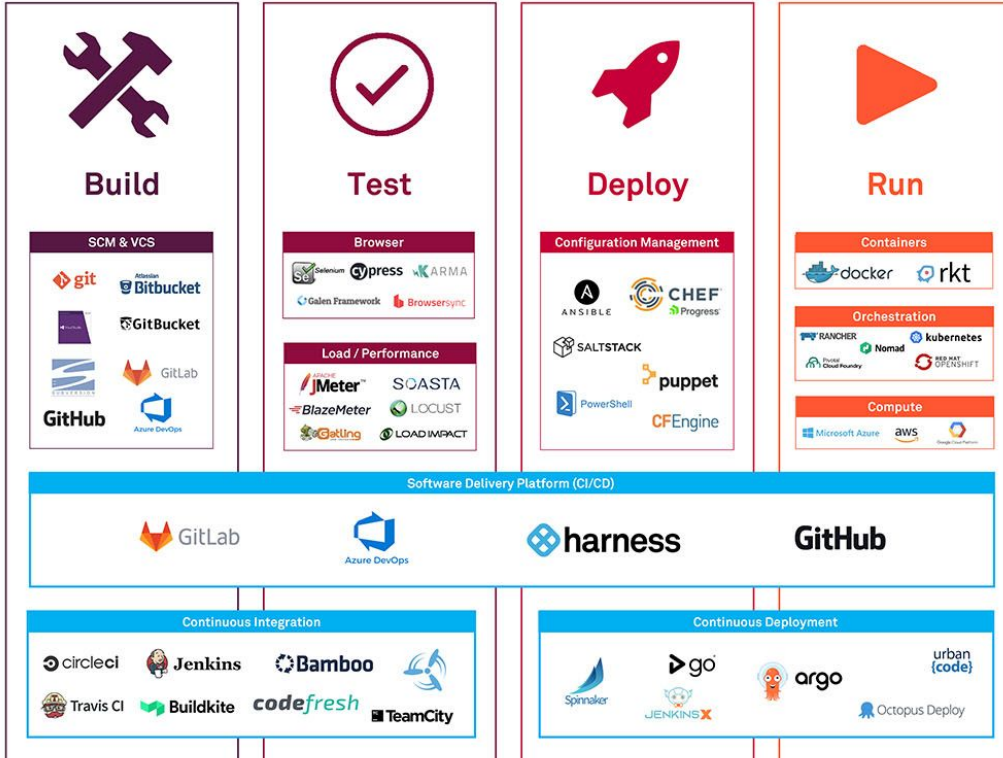
Latest unification attempt:

https://docs.dagger.io/

What we are going to look at?

- In this session we are going to go through the CI language adopted by Gitlab (baltig INFN) and the so called Github actions
- We are going to look at real life examples focused on both image building and simple application packaging
- An hands on session will focus on apply a gitlab pipeline to build the image built on the previous hands on.

References

References

- https://docs.docker.com/engine/reference/builder/
- https://github.com/opencontainers/artifacts#project-introduction-and-scope
- https://oras.land/
- https://github.com/docker-slim/docker-slim
- https://docs.gitlab.com/ee/ci/introduction/index.html#continuous-integration
- https://docs.gitlab.com/ee/ci/pipelines
- https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration
- https://github.com/marketplace?type=actions
- https://docs.docker.com/docker-hub/download-rate-limit
-