Class Announcements

1) Project teams

- Posted UMD DATA605 - Class Project Teams - Spring 2023
- If your name is not on any of the teams, please send me an email
- No midterm or final exam, complete class project to get a grade

2) Team composition

- Teams based on your self-assessed skills
- In each group there should be someone with experience with Git, Docker, Python, and so on
- Teams are not perfect, but none of your team at your future jobs will be
- Working in a team is a skill that takes a long time to hone: let's start practicing it

3) Class project complexity

- Projects have about the same complexity
- If not, we will try to account for this when grading
- Projects were assigned randomly to the teams

Class Announcements

4) Add personal info

- Fill the spreadsheet UMD DATA605 - Class Project Teams - Spring 2023 with your information
- Email
- GitHub username (free account at https://github.com)
- Telegram Handle for IM (free account at https://telegram.org)

5) Next steps

- Read carefully UMD DATA605 - Class Project
- If things are not clear send us an email or add a comment to the Google Doc
- Read the Google doc corresponding to your assigned project from UMD DATA605 - Class Project Teams - Spring 2023
- Read the first Deliverable 1 from UMD DATA605 - Class Project

6) Golden rule

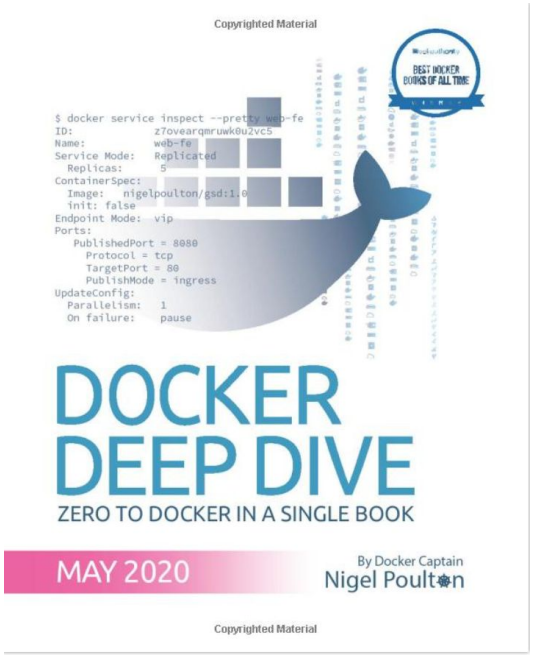
- *Treat others how you want to be treated*
  - Everybody comes from a different place and different skill level, somebody has a job, somebody has a full-time work
- *If you want to go fast, go alone; if you want to go far, go together*

UMD DATA605 - Big Data Systems

DevOps with Docker

Docker - Resources

- Concepts in the slides
- Tutorials:
  - tutorial_docker
  - tutorial_docker_compose
- We will use Docker during the project
- Web resources
  - A Beginner-Friendly Introduction to Containers, VMs and Docker
  - Official Docker Getting Started Tutorial
- Mastery:
  - Poulton, Docker Deep Dive: Zero to Docker in a single book, 2020

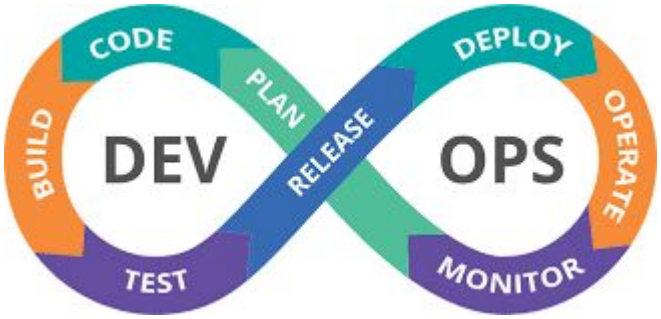


Application Deployment

- For (almost all) Internet companies, **the application is the business**
  - If the application breaks, the business stops working
  - E.g., Amazon, Google, Facebook, on-line banks, travel sites (e.g., Expedia)
- **Problem**
  - How to release / deploy / manage / monitor applications?
- **Solutions**
  - Before 2000s: “bare-metal era”
  - 2000s-2010s: “virtual machine era”
  - > ~2013: “container era”

DevOps

- **DevOps** = set of practices that combines:
  - software development (dev)
  - IT operations (ops)
- **Containers revolutionized DevOps**
  - Enable true independence between applications and IT ops
    - One team creates an application
    - Another team deploys and manages the applications
  - Create a model for better collaboration (fewer conflicts) and innovation
    - IT: “It doesn’t work!”
    - Devs: “What? It works for me”



Before Virtualization

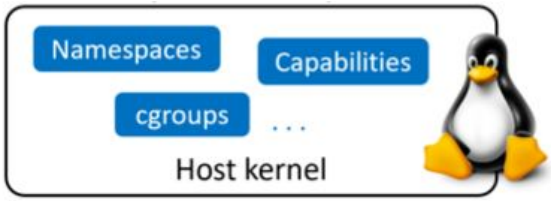
- **< 2000s: "Run on bare metal"**
  - Running one or few applications on each server without virtualization
- **Pros**
  - No virtualization overhead
- **Cons**
  - Not safe / not secure since no separation between applications
  - Expensive
- **Expensive / low efficiency**
  - IT would buy a new server for each application
  - Difficult to spec out the machine → buy "big and fast servers"
    - Overpowered servers operating at 5-10% of capacity
    - Tons of money in the 2000 DotCom boom was spent on machines (Sun Servers) and networks (Cisco)

Virtual Machine Era

- **Circa 2000-2010: Virtual Machine**
  - Virtual machine technology = run multiple copies of OSes on the same hardware
- **Pros**
  - VM runs safely and securely multiple applications on a single server
  - IT could run apps on existing servers with spare capacity
- **Cons**
  - Every VM requires an OS (waste of CPU, RAM, and disk)
  - Monitor and patch each OS
  - Buy an OS license
  - VMs are slow to boot

Containers Era

- **Circa 2013: Docker becomes ubiquitous**
- **Docker**
  - Didn't invent containers
  - Made containers simple and mainstream
- Linux supported containers for some time
  - Kernel namespaces
  - Control groups
  - Union filesystems
- **Pros**
  - Containers are fast and portable
  - Containers don't require full-blown OS
  - All containers run on a single host
  - Reduce OS licencing cost
  - Reduce overhead of OS patching and maintenance
- **Cons**
  - CPU overhead
  - Toolchain to learn / use

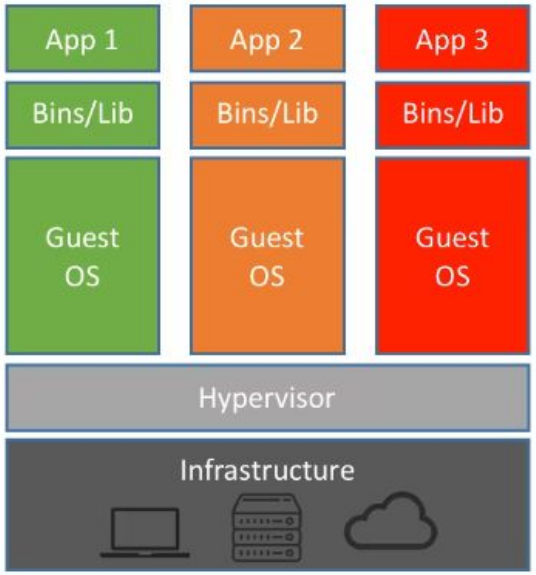


Serverless Computing

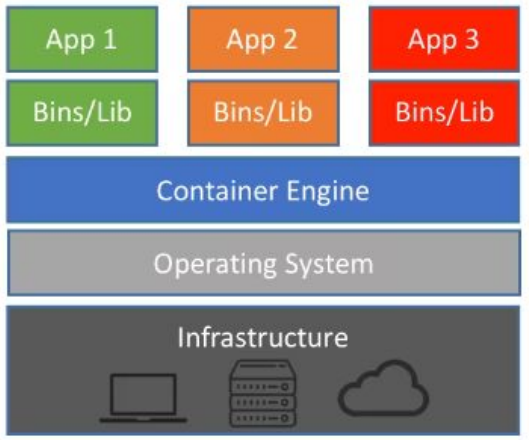
- **Containers run in an OS, OS runs on a host**
  - **Where is the host?**
    - Local (your laptop)
    - On premise (your own computer in a rack)
    - Cloud instance (e.g., AWS EC2 instance)
  - **What is the host running?**
    - Bare-metal server
    - On a virtual machine
    - On a virtual machine running a virtual machine
- **Serverless computing**
  - As long your application runs somewhere, you don't care "how" or "where"
  - E.g., AWS Lambda

HW vs OS Virtualization

- **Hypervisor performs HW virtualization**
  - Carves out physical hardware resources into VMs
  - Resources (CPUs, RAM, storage) are allocated to a VM
  - It's like having multiple computers
- "Virtual machine tax"
  - To run 3 apps, you need 3 VMs
    - Each VM requires time to start
    - Consume CPU, RAM, storage
    - VM needs a OS license
    - VM / OS need admins, patching
  - You just want to run 3 apps!
- **Containers perform OS virtualization**
  - It's like having multiple OSes



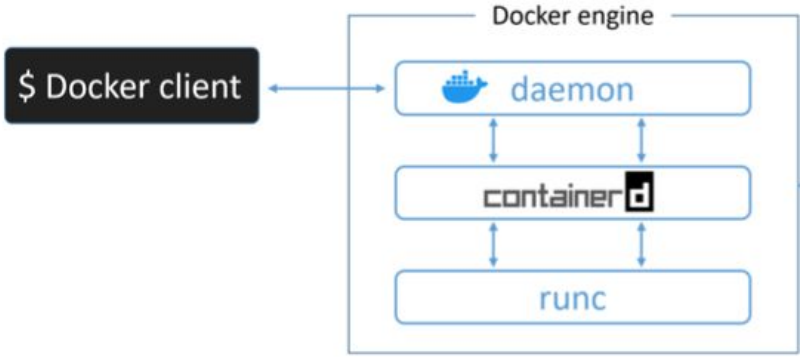
Machine Virtualization



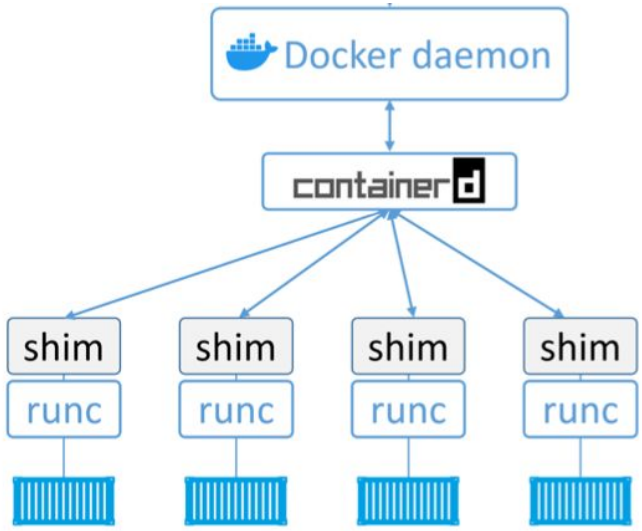
Containers

Docker: Server-Client

- Server-client architecture

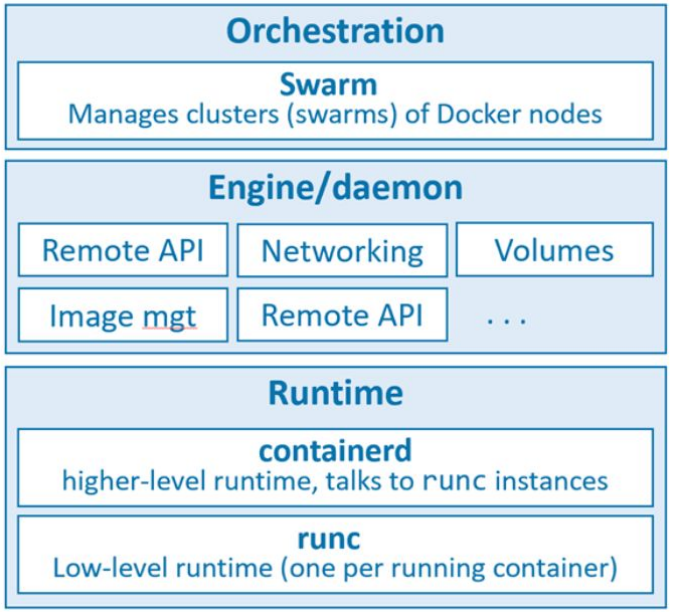


- **Docker client**
  - Command line interface
  - Communicate with the server through IPC socket (e.g., `/var/run/docker.sock`) or IP port
- **Docker engine**
  - Run and manage containers
  - Modular and built from several OCI-compliant sub-systems
    - E.g., Docker daemon, `containerd`, `runc`, plug-ins for networking and storage



Docker Architecture

- **Docker run-time**
  - `runc`: start and stop containers
  - `containerd`
    - Pull images
    - Create volumes, network interfaces
- **Docker engine**
  - `dockerd`
    - Expose remote API
    - Manage images, volumes, networks
- **Docker orchestration**
  - `docker swarm`
  - Manage clusters of nodes
  - Replaced by Kubernetes



- **Open Container Initiative (OCI)**
  - Standardize low-level components of container infrastructure
  - E.g., image format, run-time API
  - “Death” of Docker

Docker Container

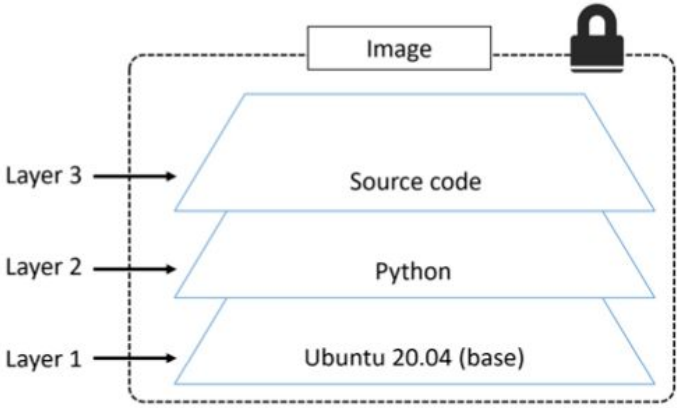
- **Docker Container**
  - Lightweight, stand-alone, executable software package
  - Include everything needed to run
    - E.g., code, runtime / system libraries, settings
  - It is a run-time object
    - vs Docker images are built-time objects
    - Like program running (container) vs program code (image)

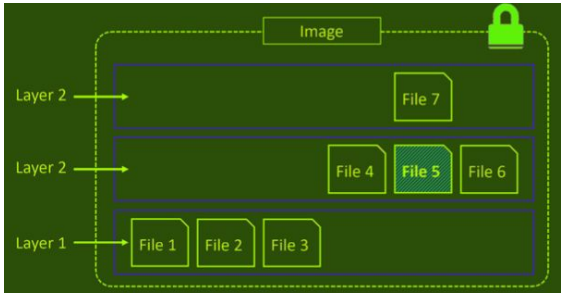
Docker Image

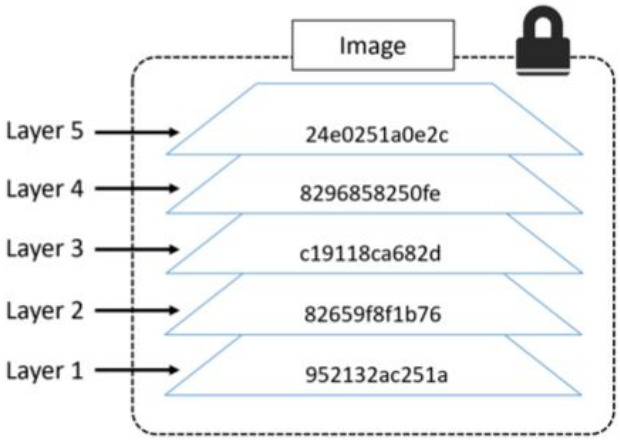
- **Docker Image**
  - Unit of deployment
  - Contain everything needed by an app to run
    - Application code
    - Application dependencies
    - Minimal OS support
    - File system
  - Users can
    - Build images from Dockerfiles
    - Pull pre-built images from a registry
  - Multiple layers stacked on top of each other
    - Typically few 100s MBs

Docker Image Layers

- **Docker image** is a configuration file that lists the layers and some metadata
  - It is composed of read-only layers
  - Each layer is independent from each other
  - Each layer comprising of many files
- **Docker driver**
  - Stacks these layers as a unified filesystem
  - Files from the top layers can obscure the files from the bottom layers
  - Implements a copy-on-write behavior
- **Layer hash**
  - Each layer has an hash based on its content
  - Layers are pulled and pushed compressed
    - A "distribution hash" is used
- **Image hash**
  - Each image has an hash
  - The hash is function of the config file and of the layers
  - When an image changes, a new hash is generated





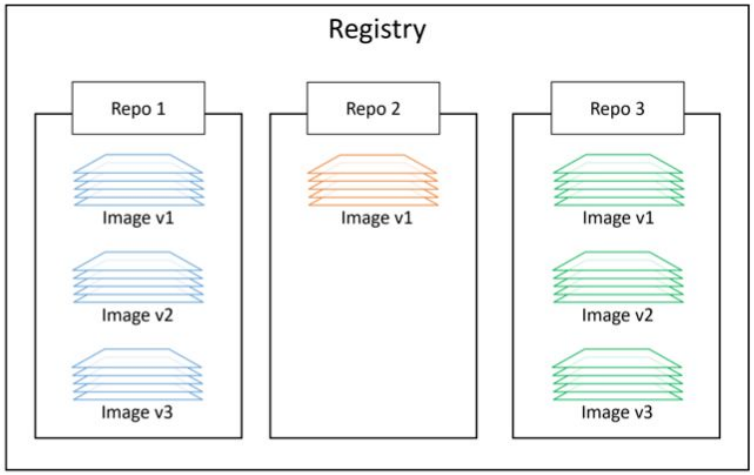


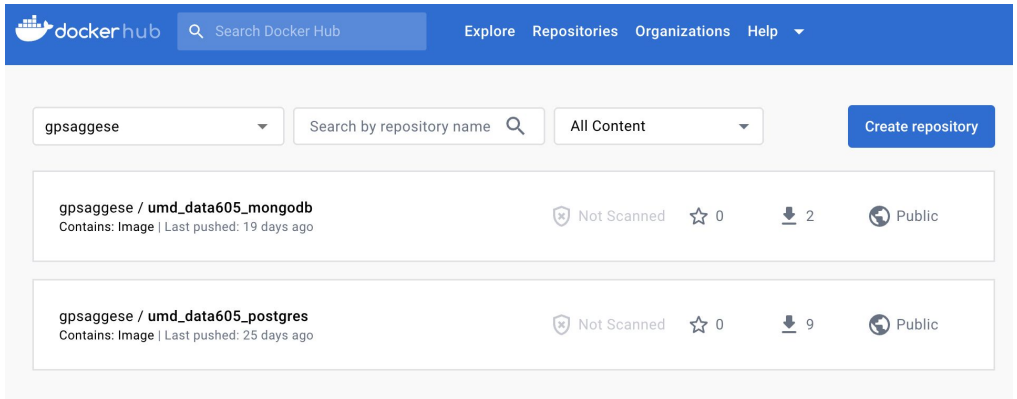
Docker: Container Data

- A container has access to different data
- **Container storage**
  - It is a copy-on-write layer in the image
  - It is ephemeral (only tmp data)
  - Data inside of containers is persisted as long as the container is not killed
    - If you stop or pause a container data is not lost
  - Containers are designed to be immutable
    - It's not good practice to write “persistent” data into containers
- **Bind-mount a local dir**
  - = a local dir is mounted to a dir inside a container
- **Docker volumes**
  - Docker provides volumes that exist separately from the container
    - E.g., to store the content of a Postgres DB
  - State is permanent across container invocations
  - Can be shared across containers

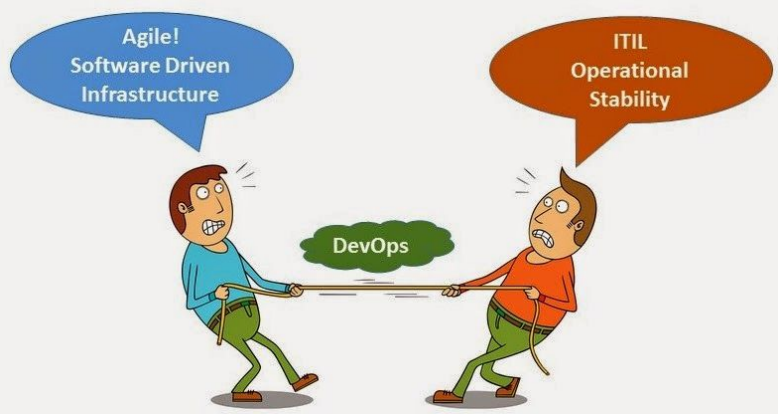
Docker Repos

- **Docker Repo**
  - Aka Docker Registry
  - Store Docker images
    - `<registry>/<repo>:<tag>`
    - `alpine:latest`
    - E.g., DockerHub, AWS ECR
  - Some repos are vetted by Docker
  - Unofficial repos shouldn't be trusted
  - E.g., https://hub.docker.com/





Devops = Devs + Ops



- **Devs**
- Implement the app
  - E.g., Python, virtual env
- Containerize the app
  - Create Dockerfile
  - Contain the instruction on how to build an image
- Build image
- Run the app as a container
- Test “locally”

- **Ops**
- Download container images
  - Contain filesystem, application, app dependencies
- Start containers
- Destroy containers
- In case of issues, it’s easy to repro the problem
  - “Here is the log”
  - Run command line
  - Deploy on a test system and debug

Containerizing an App

- = create a container with your app inside

- Develop your application code using the needed dependencies
  - E.g., install dependencies
    - Directly inside a container
    - Inside a virtual env
- Create a Dockerfile describing:
  - your app
  - its dependencies
  - how to run it
- Build image with `docker image build`
- (Optional) Push image to a Docker image registry
- Run / test container from image
- Distribute your app as a container (no installation)

Building a Container

- **Dockerfile**
  - Describe how to create a container
- **Build context**
  - `> docker build -t web:latest .`
  - Sent to Docker engine to build the application
  - Directory containing the application and its dependencies
  - Typically the Dockerfile is in the root directory of the build context

Dockerfile Example

```
FROM python:3.8-slim-buster
LABEL maintainer="gsaggese@umd.edu"

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD ["python3","-m", "flask", "run", "--host=0.0.0.0"]
```

Docker Tutorial

- tutorial_docker.md

Docker Compose

- **Manage multi-container apps running on a single node**
  - Describe the app in a single declarative configuration YAML file
    - Instead of scripts with long Docker commands
  - Compose talks to Docker API to achieve what you requested
  - E.g., you need a client app and Postgres DB
  - E.g., microservices
    - Web front-end
    - Ordering
    - Back-end DB
- In 2020 Docker Compose has become an open standard for “code-to-cloud” process

- **Manage multi-container apps running on multiple hosts**
  - Docker Stacks / Swarm
  - Kubernetes

Docker Compose: Commands

```
> docker compose --help

Usage:  docker compose [OPTIONS] COMMAND

Options:
      --env-file string            Specify an alternate environment file.
  -f, --file stringArray           Compose configuration files
  -p, --project-name string        Project name

Commands:
  build       Build or rebuild services
  convert     Converts the compose file to platform's canonical format
  cp          Copy files/folders between a service container and the local filesystem
  create      Creates containers for a service.
  down        Stop and remove containers, networks
  events      Receive real time events from containers.
  exec        Execute a command in a running container.
  images      List images used by the created containers
  kill        Force stop service containers.
  logs        View output from containers
  ls          List running compose projects
  pause       Pause services
  port        Print the public port for a port binding.
  ps          List containers
  pull        Pull service images
  push        Push service images
  restart     Restart containers
  rm          Removes stopped service containers
  run         Run a one-off command on a service.
  start       Start services
  stop        Stop services
  top         Display the running processes
  unpause     Unpause services
  up          Create and start containers
  version     Show the Docker Compose version information
```

Docker Compose: Tutorial Example

- The default name for a Compose file is `docker-compose.yml`
  - You can specify `-f` for custom filenames
- **Top-level keys** are:
  - `version`:
    - Mandatory first line to specify API version
    - Ideally always use the latest version
    - Typically 3 or higher
  - `services`:
    - Define the different microservices
  - `networks`:
    - Creates new networks
    - By default it creates a `bridge` network to connect multiple containers on the same Docker host
  - `volumes`:
    - Creates new volumes
- **Key in services** describe a different “service” in terms of container
  - **Inner keys** specify the params of Docker run command

```yaml
version: "3.8"

services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 5000
        published: 5000
    networks:
      - counter-net
    volumes:
      - type: volume
        source: counter-vol
        target: /code

  redis:
    image: "redis:alpine"
    networks:
      counter-net:

networks:
  counter-net:

volumes:
  counter-vol:
```

Docker Compose: Tutorial

- Example taken from https://github.com/nigelpoulton/counter-app

  - tutorial_docker_compose

```
> cd tutorials/tutorial_docker_compose
> vi tutorial_docker_compose.md
```