Introduction

Presentations

- Me:
  - Luca Giulianini, Cesena.
  - Linkedin: www.linkedin.com/in/luca-giulianini/
  - GitHub: www.github.com/Giulianini
- Career:
  - MSC in Computer Science -> may-2021
  - Software Engineer in Alexide -> 2021-now
- Theemes:
  - System Design, UML design, system architectures, system engineerin -> I love to design, architect and model systems.
  - Machine learning: Classic ML, Classic CV, DL, CNN, RNN, LSTM, basic TM
  - DevOps: need for automation
  - Languages: Scala, Python, Java, C#, Go, C, Bash, and web
  - IOT: MQTT, sensors, actuators, lights, protocols, ecc

DIY

- Personal projects
  - Telegram Home Control Bot: Link
  - Telegram MakeUp Bot: Link
- Bot for a particular application logic
- Open source and open to anyone who wants to contribute, also among you.
- Contributing
  - Growth in knowledge
  - Big companies really appreciate contributions
  - You can try real examples of technologies

Course Material

Code and slides

- Slides and code snippets are also pushed to repositories on GitHub

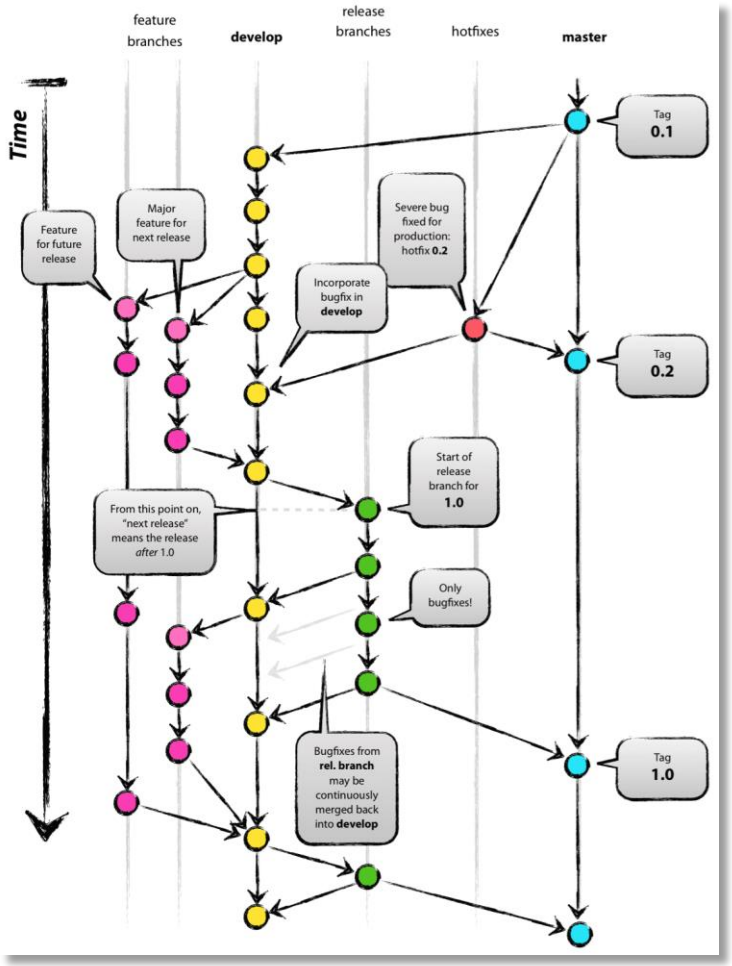
www.github.com/Giulianini/Docker101

- Make a lot of questions (make me feel the Impostor Syndrome)
- Guideline:
  - Slides with "→ Example" refer to a practical example on notebook or shell
  - All the "floating code" in slides is present in the form of script.sh inside corresponding folders. Ex docker-intro/scripts/docker-run.sh
- Examples:
  - 1 Real example of a sample project → REST API – Controller – Model (DBs)
    - Increased in complexity → to explain why we need those technologies
    - Increased in docker complexity → to explain why docker is powerful
    - Architecture is present → in the form of UML diagrams written with PlantUML

Git learn –force

Bash code – (bash, zsh on Osx/Linux | gitbash on windows)

```
mkdir project-folder                # Create a project folder
cd ./project-folder                 # Move inside the folder
# Git default settings
git config --global user.name "FIRST_NAME LAST_NAME" # Set up user name
git config --global user.email "MY_NAME@example.com" # Set up user email
# Init a repository
git init                            # Initialize a git repo in the actual folder
# Create and move to another branch
git branch [-d] feature/docker-test # Creates [delete] a branch from actual 'master'
branch
git checkout feature/docker-test    # Move the head to the branch
# or
git checkout -b feature/docker-test # Create a new branch and point the head to it
# Operations on branch
git status                          # Must become an habit/obsession/tick
git add .                           # Add all files edits in '.' to the staging area
git commit -m "message"             # Commit/Save the edits in the staing area
git logs [-a]                       # See all commits for the actual branch/all branches
git push -u origin feature/docker-test  # Push the history to remote branch first time
git push                            # Push from feature/docker-test ->
origin/feature/docker-test
git pull origin feature/docker-test # Pull from remote branch to actual branch
git pull                            # Pull from feature/docker-test ->
origin/feature/docker-test
# Merge the feature branch to main branch
git checkout master                 # Return to main branch
git merge feature/docker-test       # Merge branch to actual 'master' branch
# Clean the edits but not deleting them -> puts in a stack
git stash                           # Put the staging edits to a stack and clean all
git stash push                      # Reput the staging edits to the staging area
git stash pop
git diff                            # Shows all edits
git reset                           # Reset all edits, use with caution
```



[GitFlow]

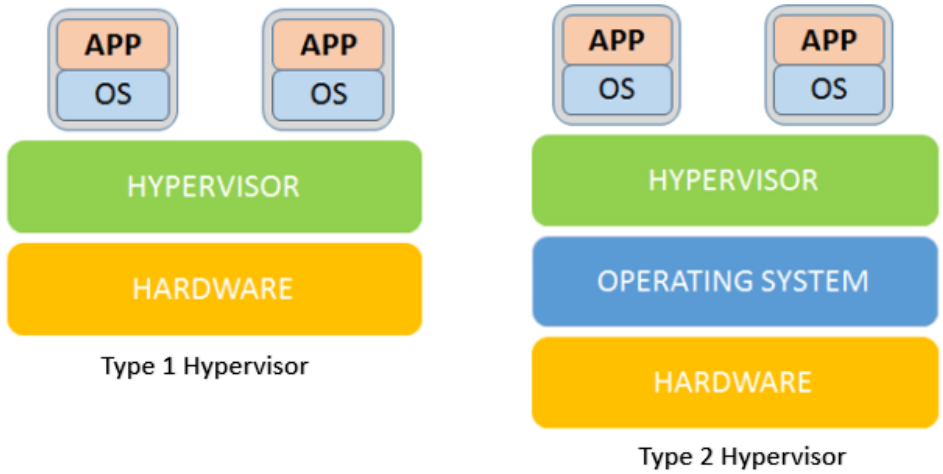
Docker101

From zero to hero

ALEXIDE SOLIDRULES

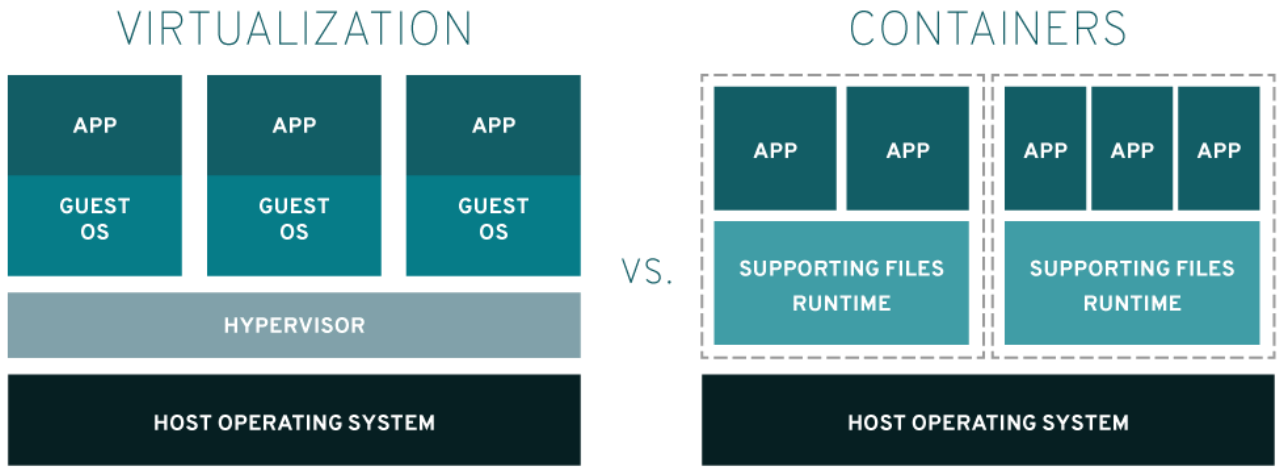
Hardware Virtualization (aka VM)

- Definition:
  - Run single/multiple **OS** from **a single Machine/OS** in a 1) secure 2) isolated **environment** called **virtual machine VM** managed by an **hypervisor** application (HV).
  - **Virtual Resources:** the hypervisor virtualizes the **hardware**
  - The machine/OS on which the vm is running is called **host**
  - The OS running in the vm is called **guest**
  - **TLDR:** the guest OS thinks to be running on the original **hardware** but it doesn't realize that's a layer (VM) between them
  - **NB:** obiously guest OS must be compatible to Host Hardware so that HV can create the layer -> if not, VM must include an **emulation** layer to convert machine code from guest to host
- Type of hypervisors
  - **Bare Metal:** hypervisor installed from scratch on a clean machine like a new OS
    - **Pros:** performances in-vm
    - **cons:** slower than a bare metal os installation
  - **Hosted:** the hypervisor is installed like a normal app
    - **Pros:** multiple os can be installed
    - **Cons:** slower than a bare metal hypervisor
- Containers???
  - The hypervisor does not virtualize the hardware, only **part of the OS**



Partition Virtualization – (aka Containers)

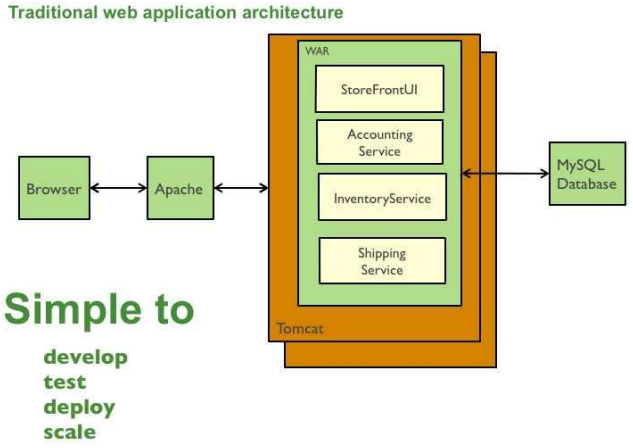
- Definition:
  - A container runs on the **same kernel** as the host OS. It's an instance of a **user-space** though it provides **isolation** from: 1) host OS 2) other containers.
  - **Virtual Resources**: the hypervisor virtualizes an **OS Partition** called Container
  - Built on Linux cgroups and namespaces
  - **Isolation:** filesystem, IPC and network are isolated
  - **Management:** CPU, Network, Filesystem
- Advantages:
  - **Fast and Lightweight**:
    - **Lower overhead:** relaying on host OS features
    - **Lower memory:** no full virtualization like VM
  - **Secure:** isolation between containers
  - **Portable:**
    - **CI/CD:** simplify build/test on specific environments
    - **Clean environment:** no more junk from stupid application installers. Deleting a container delete everything inside it
  - **Microservices:** decoupling applications into small independent/resilient pieces
  - **IT'S ALL ABOUT Deployment**: application packaging -> portable image -> automatic deployment -> on a secure and lightweight container -> replication/scaling

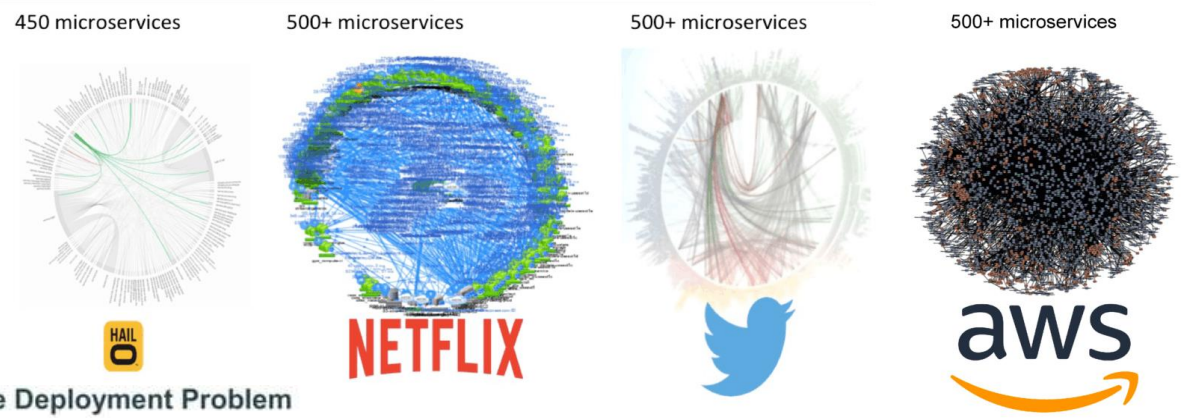


1) **Namespaces:** are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources

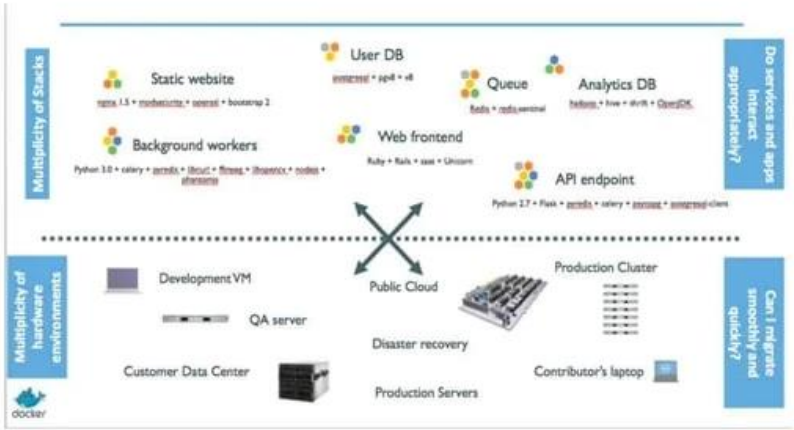
Software industry has changed

- Before:
  - Monolithic applications
  - Working on single environment
  - Scaling vertically (CPU cores, memory)
  - Deployment consists of a CD or an EXE published on the web
  - Developed using waterfalls cycles
- Now:
  - Monoliths are now exploded into sub services: Microservices
  - **Microservice:**
    - Has a **delimited scope**
    - Probably written by a different team
    - **Network** communication between services
    - Has to scale horizontally (more machines)
  - Deployment is heterogeneous:
    - **Different targets**: Cloud, On-Premise
    - **Different tech stack**
  - PM Cycle are smaller and faster:
    - iterative models like SCRUM, KANBAN
    - Industry is more competitive



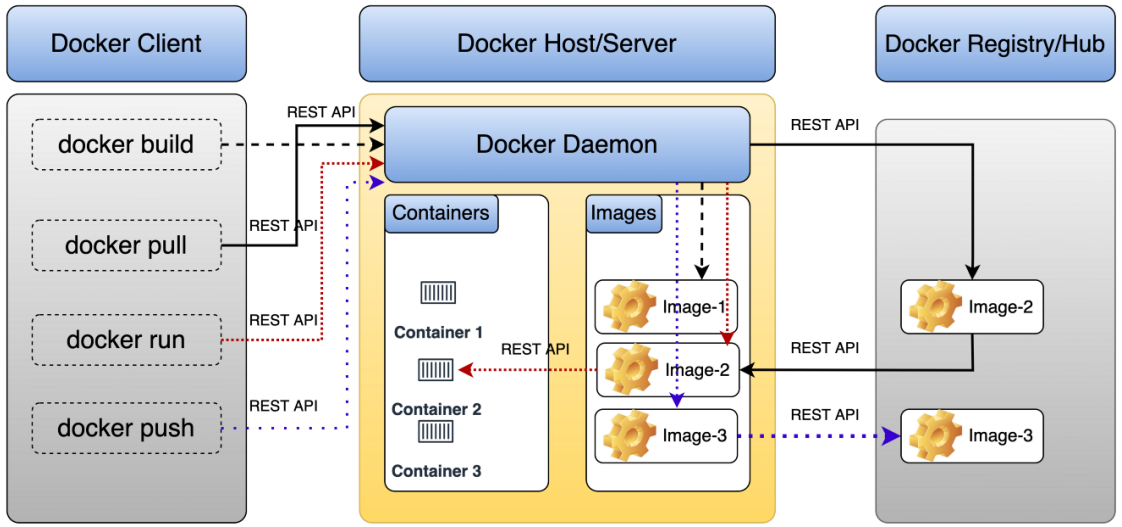


The Deployment Problem



Docker

- Developed in 2013 – based on a Linux layer called **Linux Container (LXC)**
- Written in Go
- Products:
  - **Docker Engine**: a daemon which manages linux containers, the same way hypervisor manages VMs. Interfaces with underlying Linux OS and managed lifecycle of containers through simple API that can be called from outside.
  - **Docker CLI**: interface to interact with the docker engine though commands exchanged the API exposed by the Docker Engine
  - **Docker Desktop**: gui interface to interact with docker engine
  - **Docker Enterprise**: you will see it in your future company
  - **Docker Registry/Hub**: docker images are stored here
- Abstractions
  - **Docker client**: the CLI/GUI that communicate to server/host
  - **Docker Hub/Registry:** where images are stored
  - **Docker Host/Server**: the docker core engine:
    - **Image:** a packaged environment behave like a "stampino" for containers
    - **Container:** an image instance. Like objects and classes in OOP
    - **File System:** can be mapped/shared by containers
    - **Network:** can be mapped/shared by to containers



Docker simple container operations

Simple run (name and interactive) + container lifecycle

```
# General status
docker info                 # Docker engine core info
docker version              # Check if engine and cli are installed

# SIMPLE create and run a container from a remote image
docker search ubuntu        # Search for the image ubuntu. Check for official status, more secure
> NAME                      DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
> ubuntu                    Ubuntu is a Debian-based Linux operating sys…   15159     [OK]
docker pull ubuntu          # Download the Ubuntu image (stampino)
> docker.io/library/ubuntu:latest
docker run ubuntu           # Create an instance (container) from the ubuntu image with a random name
docker run -it ubuntu       # Like above but attaches to STDIN
docker run -it --name="ubuntu_cont“ ubuntu # Like above but with a container name
> root@4807e7c6f422:/#

# Manage images (we will see how to create images with Dockerfile)
docker images               # Show all images
docker rmi ubuntu           # Delete the ubuntu image

# Manage containers lifecyle
CTRL+P+Q                    # Detach from the container
docker ps [-a]              # Manage active [all] containers
> CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS         PORTS     NAMES
> 4807e7c6f422   ubuntu    "bash"    3 seconds ago   Up 3 seconds             hopeful_bhabha
> 4efeff2c7d13   ubuntu    "bash"    4 seconds ago   Up 3 seconds             ubuntu_cont
docker pause ubuntu_cont    # Pause the container
docker unpause ubuntu_cont  # Unpause the container
docker stop ubuntu_cont     # Stops the container with name="ubuntu_cont"
docker stop 4efeff2c7d13    # Stops the container with id="4efeff2c7d13"
docker start ubuntu_cont    # Starts the container
docker restart ubuntu_cont  # Restarts the container
docker rm ubuntu_cont       # Delete the container
```

Docker complex container operations

Complex run (remove, background, attach, detach, exec)

```
# Create a container and run commands inside
docker run -it --name="ubuntu_cont" ubuntu  # Create a new container
root@477f547a8618:/# apt-get update && apt install -y net-tools
root@477f547a8618:/# ifconfig
root@477f547a8618:/# exit
docker ps -a
> CONTAINER ID   IMAGE     COMMAND   CREATED              STATUS                      PORTS     NAMES
> 477f547a8618   ubuntu    "bash"    About a minute ago   Exited (0) 2 seconds ago              ubuntu_cont
docker run -it --name="ubuntu_cont2" ubuntu     # Creates a new instance
root@477f547a8618:/# ifconfig                   # Not found
docker start ubuntu_cont                        # Starting ubuntu_cont restore the container state

# Run commands
docker run -it --rm --name="ubuntu_cont" ubuntu    # Creates a new instance and remove when it stops
docker run -it --rm --name="ubuntu_cont" -e "PIPPO=PLUTO" ubuntu # Init a variable
root@bfc149fdbbf3:/# echo $PIPPO
> PLUTO
docker run -it -d --rm --name="ubuntu_cont" ubuntu # Run in background like CTRL+P+Q
docker exec -it ubuntu_cont echo "hello"        # Exec a command inside a background container
> hello
docker exec -it -d ubuntu_cont echo "hello"     # Exec a background command inside a background container
docker attach ubuntu_cont                       # Attach to the container, move it in foreground
root@7904cdb7a55a:/#
CTRL+P+Q                                        # Detach
```

Docker Networks

Ports and networks

```
# Networks
docker run -it --rm --network="bridge" --name="ubuntu_cont" ubuntu  # Use default bridge
root@ace3a0bf4cc9:/# apt-get update && apt install net-tools
root@ace3a0bf4cc9:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
docker run -it --rm --network="host" --name="ubuntu_cont" ubuntu    # Use host network
root@ace3a0bf4cc9:/# apt-get update && apt install net-tools
root@ace3a0bf4cc9:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.65.3  netmask 255.255.255.0  broadcast 192.168.65.15

# Run commands
docker run -it --rm -p 8080:80 --name="ubuntu_cont" ubuntu            # Mapping 8080 to cont 80
> CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS         PORTS                   NAMES
> 8e404e12b00b   ubuntu    "bash"    11 seconds ago   Up 8 seconds   0.0.0.0:8080->80/tcp    ubuntu_cont
```

Docker Volumes

Mapping folders

```
# Mapping volumes
docker run -it --rm -v /mnt/c/Users/LUCA.GIULIANINI/Desktop:/Desktop --name="ubuntu_cont" ubuntu    # Mounting a folder
root@797ec677545b:/# ls
> Desktop  boot  etc   lib    lib64   media  opt   root  sbin  sys  usr
> bin      dev   home  lib32  libx32  mnt    proc  run   srv   tmp  var
```

Docker images

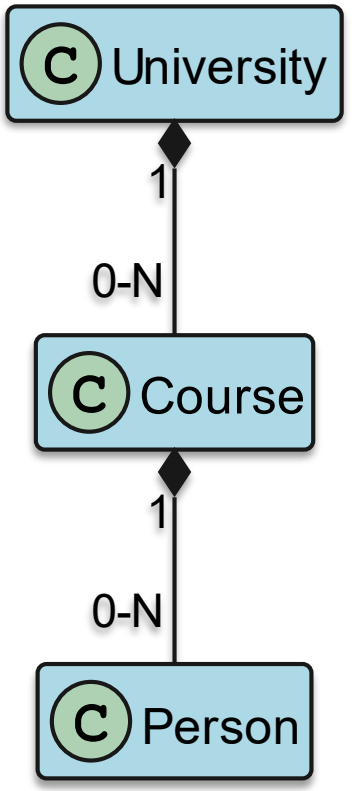
A “stampino” for docker containers. Dockerfile is a recipe to create a new image from scratch

```
# Dockerfile keywords
FROM            # Base image for building the new image
MAINTAINER      # Name of maintainer
RUN             # Command to be executed during image creation
COPY            # Copy from host to the image
ADD             # Same as COPY but with URL option
ENV             # Set a local environment inside future container
CMD             # Command to be executed when container is CREATED form image
ENTRYPOINT      # Similar to CMD, command executed when container is RUNNING
WORKDIR         # The working directory for the container when created

# Rest api Dockerfile
FROM python:3.10
WORKDIR /app
COPY . /app
RUN pip install flask
ENV FLASK_APP "app"
WORKDIR /app/restapi
ENTRYPOINT flask run

# Dockerfile commands
docker build -t rest-api-server_image . # Build a new image called 'rest-api-server_image' using Dockerfile inside '.' folder.
docker images                           # List images
docker rmi rest-api-server_image        # Delete the image created with Dockerfile
docker run -it --rm -p 5000:80 -d --name rest-api-server_container rest-api-server_image # Create a container from the image
docker image prune --all                # Remove images without associated container, very powerful
```

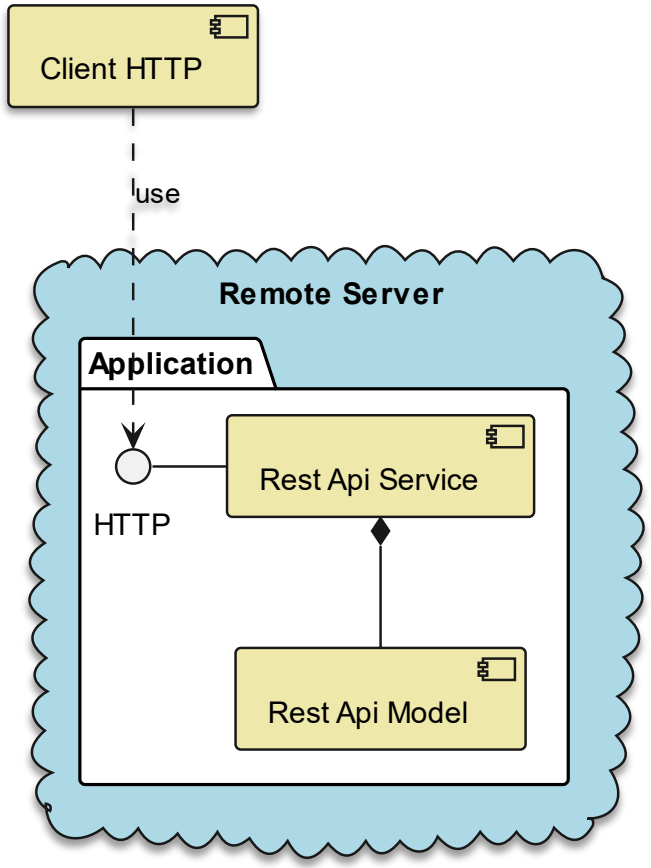
Rest Api Server → Example

- Features
  - **Rest API** endpoint made with Flask
  - Model is an **in-memory** data structure (not a DB)
  - Definition of HTTP methods for request routes
- Problems
  - Technical problems:
    - There's **no persistence in data.**
      Data is not stored safely. It's an online memory, stopping the service clear the memory
  - Deployment problems:
    - Specific python version required?
    - Need to install Flask
    - Need environment variables to be defined?
    - Configuration script required?
    - Need to backup all the data easily?
  - NB some of there are pretexts to convince you that Docker is our God.
    Indeed yes, Docker will be enough to solve most of your deployment problems but at the only condition to not overuse it!

Data Model UML



Component Diagram



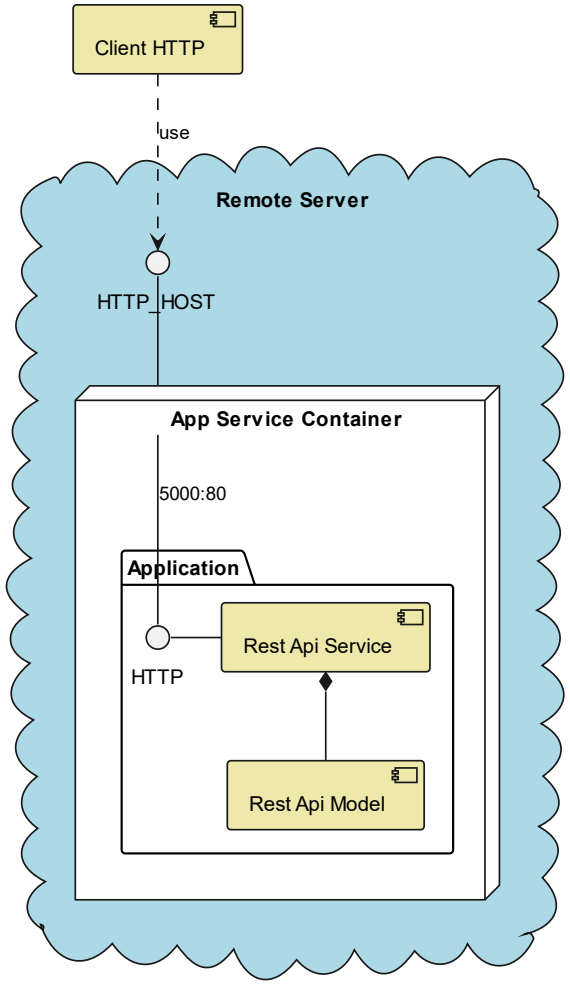
Assembling all – Docker Compose

- We know:
  - how to **pull, push and create a new image** from a basic image – Dockerfile → Automated
  - how to **create/run a container from an image with configuration** – Docker commands → NEED AUTOMATION
  - how to **manage** (start, stop, rm, attach, exec, …) it – Docker lifecycle commands → NEED A REMAPPING (on services)
- We need:
  - to put together these pieces and automate everything → **Docker Compose**
- **Compose File**
  - Define the concept of a service: a **packaged/dockerized** application created from an **image with Dockerfile** and **deployed** in a "virtual environment" (**container**) all done automagically.
  - A bit messy in concepts:
    - **Images** and **container** configuration are **mixed**
    - Need to create at least one to understand it
  - Think about it as a:
    - **Mapping** of the standard docker commands in a YAML config fashion → volumes, IP, background
    - A **definition** of **images** and **Dockerfile** in a YAML config → image name, docker build
    - A **general configuration** for a docker container → Network, volumes, restarting policies
- **Compose Commands**
  - **Remapping** of the standard **lifecycle** commands prefixed by `docker-compose`
  - **Definition** of a bunch of new simple commands `up/down`

Dockerizing the Rest Api Server → Example

Docker Compose is external because it must collect multiple services together

```yaml
version: "3.9"                              # The compose file version
services:                                   # Services section
  app-service:                              # Single service section
  # Image creation
    image: "rest-api_image"                 # Name of the image to create
    build:                                  # How to build the image
      context: rest-api-server              # Folder in which Dockerfile is contained
      dockerfile: Dockerfile                # Can leave empty if filename is Dockerfile
    # Container Creation - Running
    container_name: "rest-api_container"    # Name of the container to create
    restart: "unless-stopped"               # Can be always, on-failure, unless-stopped
    # networks:                             # Can define networks
    #   - my-network
    ports:                                  # Port mapping
      - 5000:80
    volumes:                                # Volume mapping
      - ./rest-api-server/shared-volume:/shared-volume

# COMMANDS FOR MANAGING SERVICES (more than container)
docker-compose config                       # Check compose file correctness
docker-compose up -d --build                # Build image, create container from image in
background, run
docker-compose up -d --build service_name   # Up single service
docker-compose logs                         # Get log from services
docker-compose ps                           # List services containers
docker-compose images                       # List services images
docker-compose down                         # Stop the services
docker-compose down -v                      # Stop and delete volume
docker-compose restart                      # Restart the services
```



Docker final form

- The docker final form is obviously Docker Compose
- Compose is container on steroids and now we will use only Compose to do anything.
- I lie there are more final form of docker like Swarm and Kubernetes but we can't cover them. You should have seen them

Some interesting (dockerized) projects

- **Vaultwarden:** bitwarden pro self-hosted, dockerized and free
- **Home Assistant:** home automation project dockerized
- **Telegram home control bot:** bot for home automation
- **PiHole:** adblocker, DNS, DHCP server
- **PiVPN:** VPN free (OpenVPN, Wireguard) dockerized