Image Classifier App - Part 3

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

Thanks to Maura Pintor for kindly allowing the reuse of her material.

Introduction to Containerization and Docker

Containerization

- Encapsulates application code, runtime, libraries, and dependencies into a single **isolated** unit: the *container*
- Runs containers on shared OS kernel
- Provides portability and predictable behavior across environments

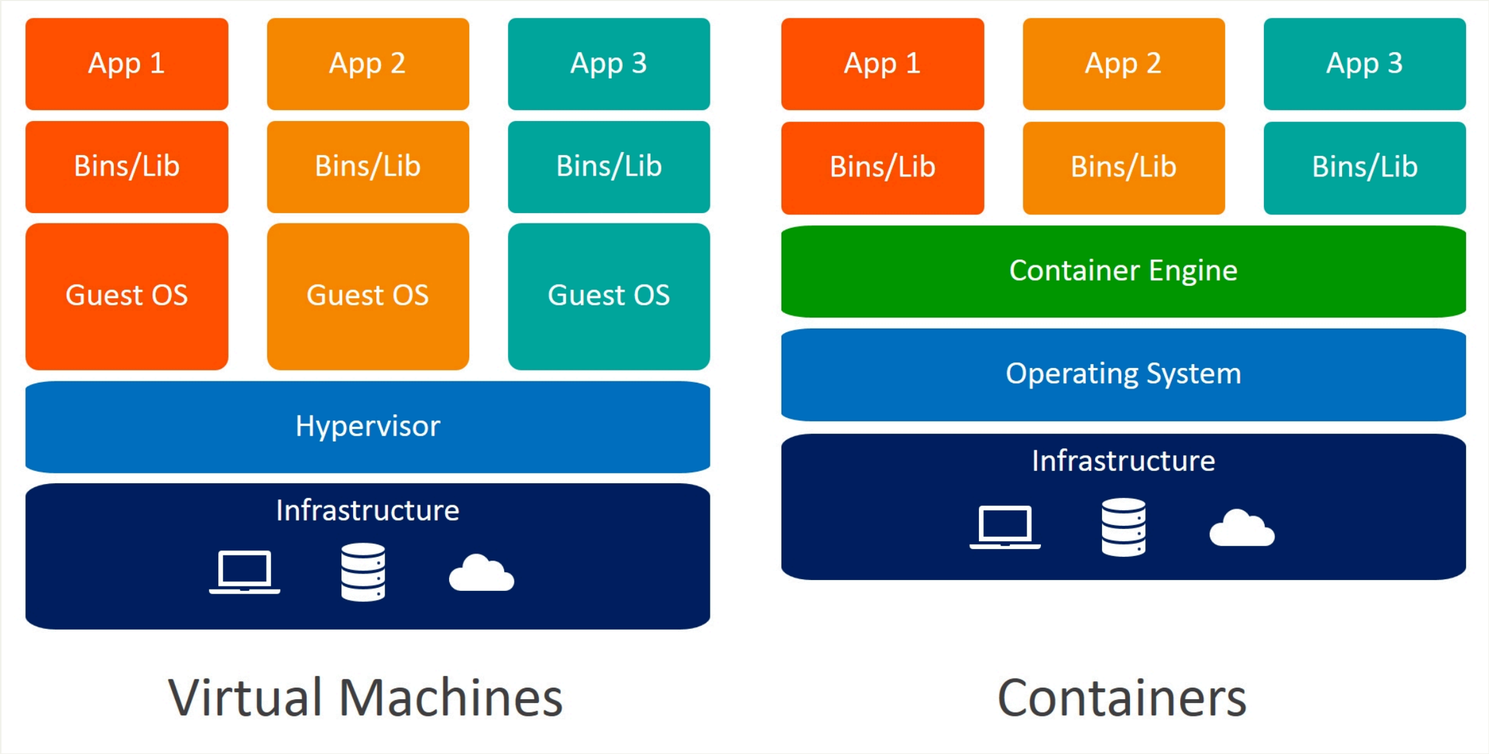
Containerization

- Encapsulates application code, runtime, libraries, and dependencies into a single **isolated** unit: the *container*
- Runs containers on shared OS kernel
- Provides portability and predictable behavior across environments

Why containers are useful:

- Traditional deployments suffer from dependency conflicts
- Virtual machines solve isolation but are heavy and slow
- Containers combine isolation with lightweight resource usage

Containers vs Virtual Machines



Key advantages of containers:

- Startup in milliseconds vs minutes for VMs
- Smaller footprint (MBs vs GBs)
- Easier scaling and orchestration
- Immutable infrastructure pattern

Where containers are typically used:

- Microservices architecture: independent deployable units
- CI/CD pipelines: consistent build and deploy
- Cloud-native: portable across providers
- Cost optimization: better density on hosts

Frameworks Overview

- Docker: most popular
- Podman: daemonless, rootless
- containerd: core runtime
- LXC/LXD: Linux containers

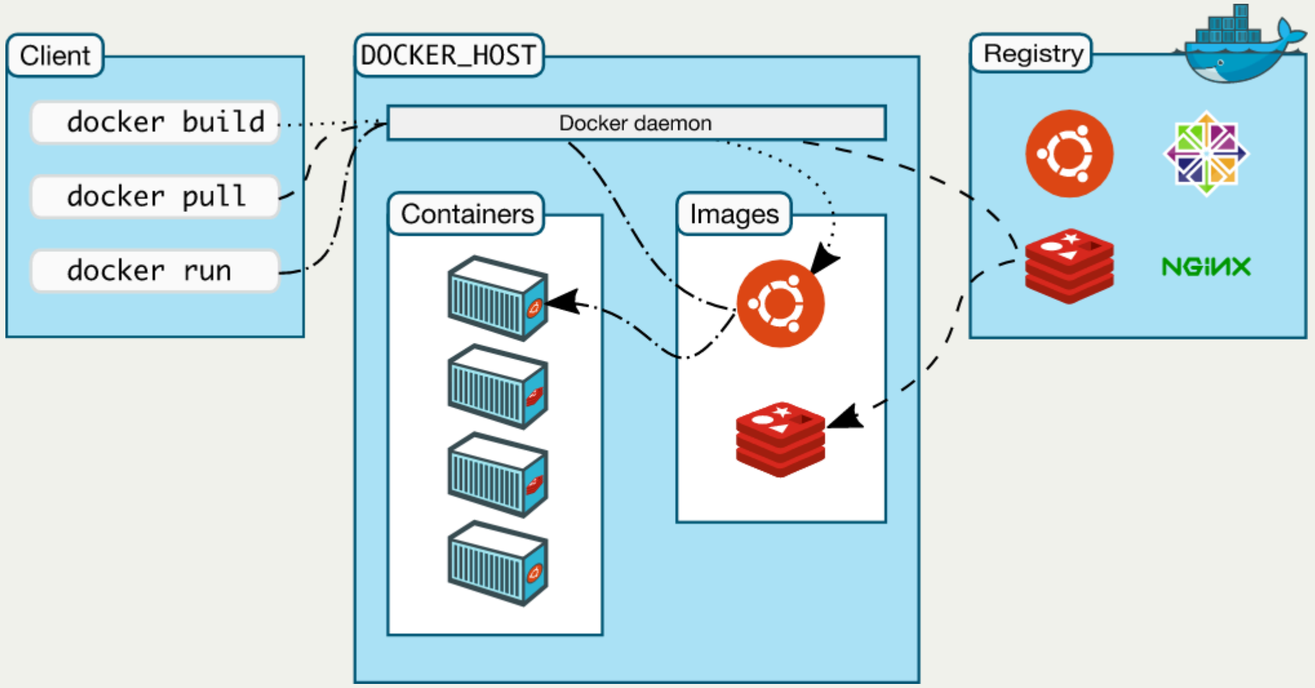
Orchestration

- Automated management of container lifecycles across clusters
- Handles scheduling, scaling, networking, health checks, and failover

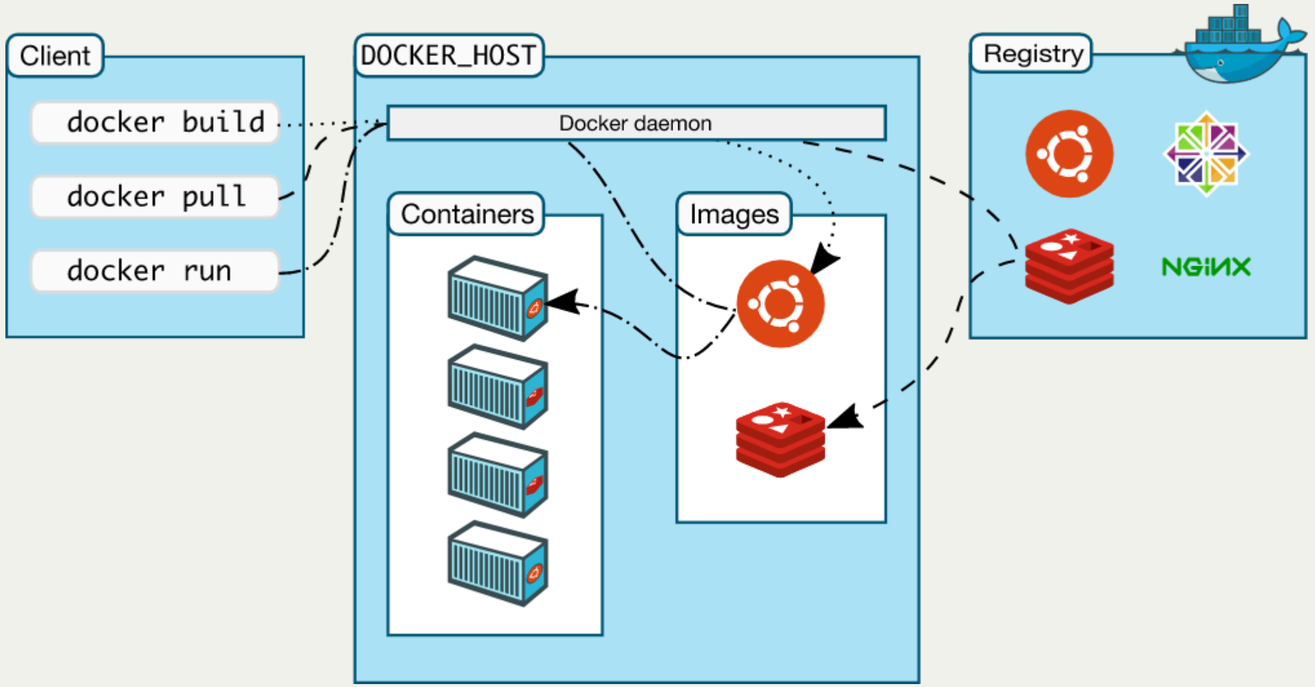
Orchestration Tools

- Kubernetes: industry standard
- Docker Swarm: simpler, Docker-native
- Nomad: lightweight alternative

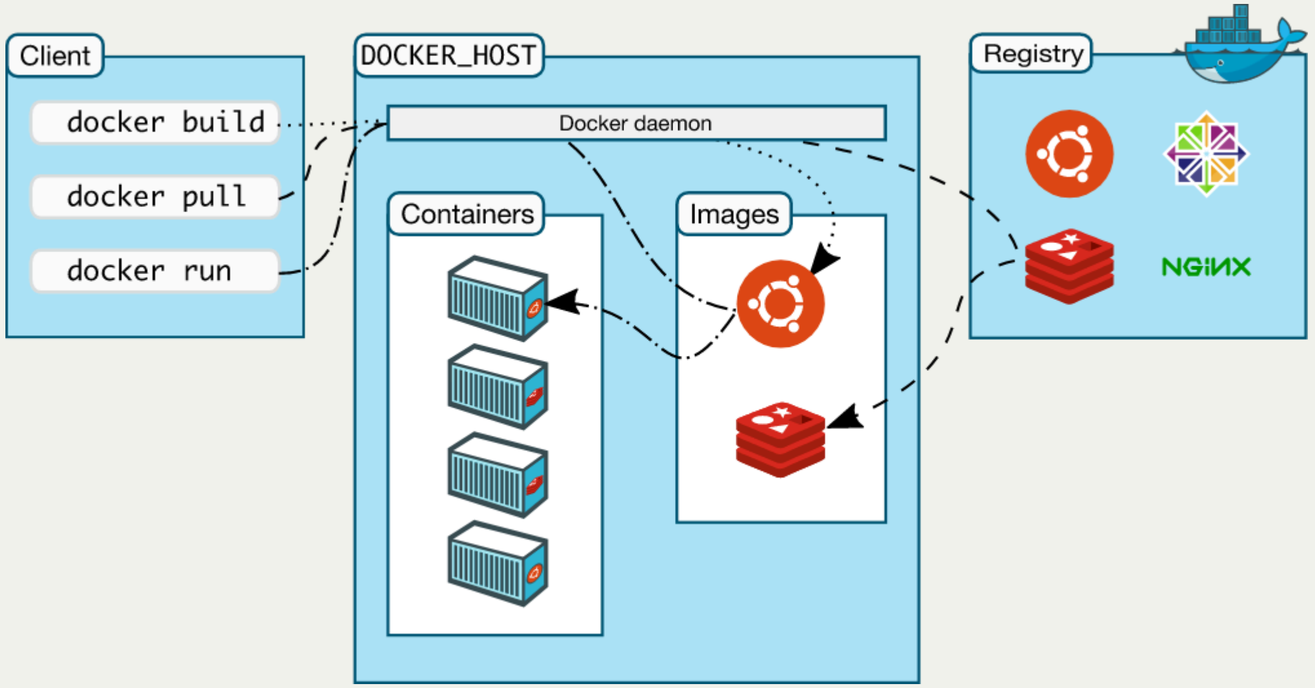
Docker



- Platform for building, shipping, and running containers
- Provides CLI, API, and daemon for lifecycle management



- **Client**: CLI and SDK for user interaction
- **Daemon**: core engine managing containers, images, volumes, networks
- **Registry**: remote image storage (Docker Hub, private registries)



- Client sends commands via REST API
- Daemon pulls images, sets up filesystem, networking, and starts containers
- Uses container runtime (runc) for low-level operations

Images vs Containers

An **image** is a static, immutable template containing all the required dependencies, libraries and application's code (including a file system).

Images vs Containers

An **image** is a static, immutable template containing all the required dependencies, libraries and application's code (including a file system).

A **container** is a executable instance of an image, running the application.

Image building

For creating a container with Docker, we need first to build the **image**.

The image specifications are written in a `Dockerfile`

- The process starts from a *base image*. Several pre-built base images are hosted in the Docker Hub
- Each instruction creates a new layer

Most common Dockerfile instructions

`FROM <image>`

- Defines a base for your image

`RUN <command>`

- Executes any commands in a new layer on top of the current image and commits the result. `RUN` also has a shell form for running commands

`WORKDIR <directory>`

- Sets the working directory for any `RUN`, `CMD`, `COPY`, and `ADD` instructions that follow it in the Dockerfile

`COPY <src> <dest>`

- Copies new files or directories from and adds them to the filesystem of the container at the path
- Alternatively, `ADD` also works with remote files URLs and compressed archives (that are automatically extracted)

`CMD <command>`

- Lets you define the default program that is run once you start the container based on this image. Each Dockerfile only has one `CMD`, and only the last `CMD` instance is respected when multiple exist

```dockerfile
FROM python:3.12

# We copy just the requirements.txt first to leverage
# Docker cache
COPY ./requirements.txt ./requirements.txt


RUN pip install -r requirements.txt

COPY . ./

CMD ["fastapi", "run", "main.py", "--port", "8000"]
```

The important part here is to remember that:

- we are starting from an image that already contains Python and some other useful tools, *e.g.*, `pip`
- we are leveraging Docker cache

Docker cache

Docker builds **intermediate containers** for every line we have in this Dockerfile.

If we change the content of one of the lines, Docker uses the cached version of everything before the changed line and rebuilds what comes after the line.

- Cache hit if instruction and context unchanged
- Cache miss invalidates subsequent layers
- Optimize by ordering stable steps first

Building the container

Now let's open a terminal in the root directory, and run:

```
docker build . -t classification-ws
```

We are telling Docker to build the current directory, and to **tag** the image we just created with the name `classification-ws`.

Docker will automatically search for a file `Dockerfile` in the specified directory.

To see all the images:

```
docker images
```

If you haven't done it yet, remember to stop the webserver that we were using until now.
This is because we will run the same service through the docker container now!

We can now run the container with the command:

```
docker run -p 80:8000 classification-ws
```

Note that we are specifying here a **socket**. This is a mapping of a port inside the container with a port in our computer.

So, inside our container we will run the server on port 8000, which will be linked with the port 80 of our localhost.

We can see that the container is running the server in our localhost port 80[4].

To see all the running containers (in another terminal):

```
docker ps
```

You can also add `-a` to see all the created containers

---

[4] note that port 80 is the default for the web browser

To remove all the stopped containers:

```
docker container prune
```

Or you can add `--rm` to automatically remove a container when it stops:

```
docker run -p 80:8000 --rm classification-ws
```

Don't run classifications there yet... What happens when we click on "submit"?

We should improve our docker service a little bit, right now it is missing the redis service, the volume, and the worker.

If we ask for a classification job now, we will be stuck with an error...

Stop the container with Ctrl+C.

We can try to make it work by replacing the `classifications` APIs with the older version without queues:

```python
@app.post("/classifications", response_class=HTMLResponse)
async def request_classification(
    request: Request,
    model_id: Annotated[str, Form()],
    image_id: Annotated[str, Form()]
):
    classification_scores = classify_image(
        model_id=model_id,
        img_id=image_id
    )
    data = {
        "model_id": model_id,
        "image_id": image_id,
        "classification_scores": classification_scores
    }
    return templates.TemplateResponse(
        request, "classification_output.html", {"data": data}
    )
```

Remember to build again the image before running again the container!!!

Otherwise, it will still contain the old source code...

...but now we can see how useful the build caching: only the last two Dockerfile instructions are executed.

Storage

By default, containers are isolated from the host filesystem for security and portability.

- Keeps containers lightweight and portable.
- Prevents accidental or malicious changes to the host.

There are two options for sharing files and directories across the host filesystem and the container.

Volumes

Persistent data stores for containers, created and managed by Docker, stored within a directory on the Docker host.

- You cannot access the volume content from the host if the volume is not mounted into a container.

Main features:

- High-performance I/O
- Not dependent on the host filesystem
- Easier to back up, migrate and managed
- Do not increase the size of the container
- Can be safely shared among multiple containers
- Not a good choice if you need to access the files from the host, as they are completely managed by Docker

Bind mounts

A file or directory on the host machine is mounted from the host into a container.

- They have write access to files on the host by default
- Containers with bind mounts are strongly tied to the host

Useful for:

- sharing source code or build artifacts between a development environment on the Docker host and a container
- create or generate files in a container and persist the files onto the host's filesystem
- sharing configuration files from the host machine to containers.

Bind mounts are also available for builds

- Mount host path into container
- Reflects real-time changes
- Ideal for development

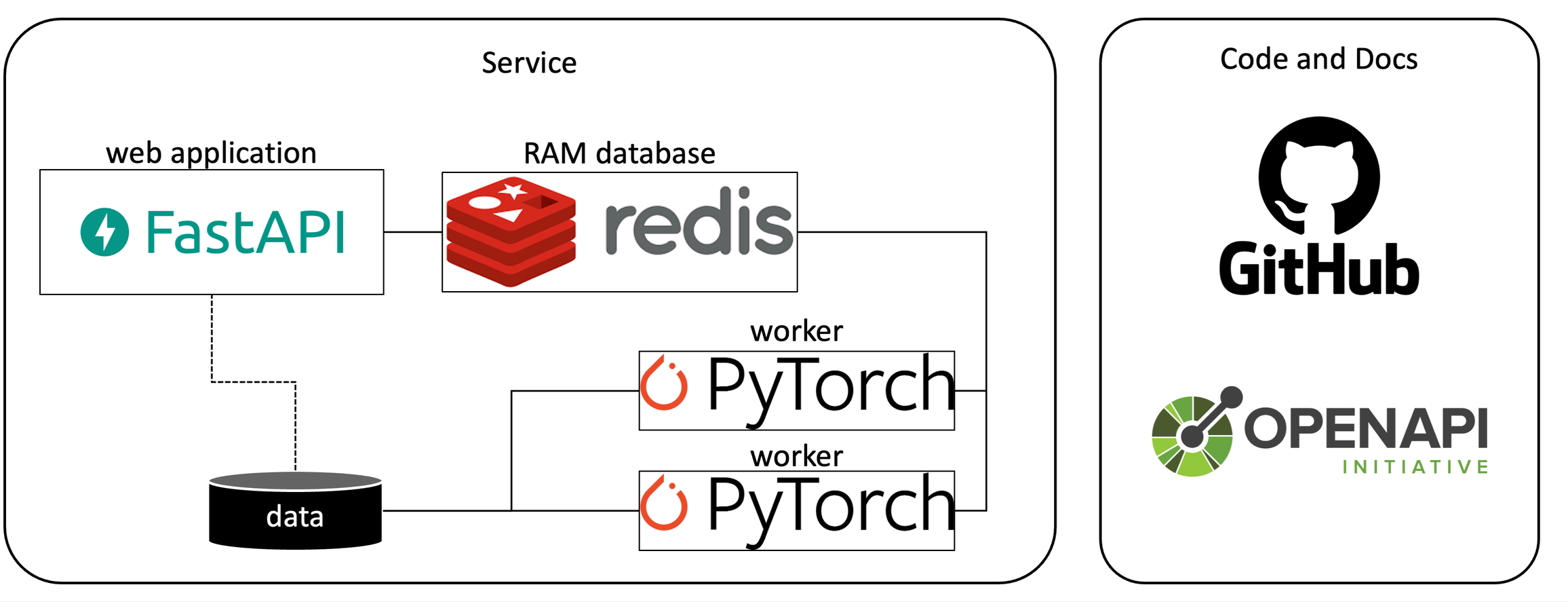
We want to leverage Docker bind mounts to share the pre-trained PyTorch models stored on our device with the container, to avoid downloading models every time we run the container.

```
docker run -p 80:8000 --mount type=bind,src=$HOME/.cache/torch,dst=/root/.cache/torch classification-ws
```

If you are not sure, always use the `ro` option!

Docker compose

Remember the architecture? See how many containers are there:



See how many containers are there now...

- one for the web application
- one for each worker
- one for the redis database

Instead of defining each container and linking them together manually, we can use the **Docker Compose** tool:

- we only need a single YAML file (the *Compose* file)
- then we can create, start, and stop everything with a single command

Services

Services are abstract resources running within containers

- They are defined by a Docker image and other runtime arguments
- They can be replicated to make the application *scale*

A Compose file must declare a `services` top-level element

- its key fields are the service names
- their corresponding values are the service definitions (e.g., the `build` section defines how to create the service Docker image)

Web application service

```
services:
  web:
    build: .
    command: fastapi run main.py --port 8000
    ports:
      - "80:8000"
    links:
      - redisdb
    environment:
      - REDIS_HOST=redisdb
      - REDIS_PORT=6378
    volumes:
      - ~/.cache/torch:/root/.cache/torch
```

The `web` container is running the webserver on port 8000.

`links` defines the connection of this container with others defined in the same Compose file.

We are connecting this container with the one running the database.

`environment` defines environment variables, that we can use for storing dynamic values like the redis port and the hostname.

Why is the hostname `redisdb`?

By default, docker links define an entry in the hosts file of our containers that points to the linked containers.
So if we connect to `resdisdb` from inside of our `web` container, we will see the localhost of the `redisdb` container.

Redis DB service

```
  redisdb:
    image: "redis"
    command: --port 6378
    ports:
      - "6378:6378"
```

Worker service

```
worker:
  build: .
  command: python -m worker
  links:
    - redisdb
  environment:
    - REDIS_HOST=redisdb
    - REDIS_PORT=6378
  volumes:
    - ~/.cache/torch:/root/.cache/torch
```

Remember to replace the `classifications` API function with the one using the queue!

```python
@app.post("/classifications", response_class=HTMLResponse)
async def request_classification(
    request: Request,
    model_id: Annotated[str, Form()],
    image_id: Annotated[str, Form()]
):
    redis_conn = Redis(Configuration.REDIS_HOST, Configuration.REDIS_PORT)
    q = Queue(name=Configuration.QUEUE, connection=redis_conn)
    task = q.enqueue(classify_image, model_id=model_id, img_id=image_id)
    return templates.TemplateResponse(
        request, "classification_output_queue.html",
        {"image_id": image_id, "jobID": task.id}
    )
```

And finally, let the magic happen! We can create our architecture with a single line:

```
docker compose build && docker compose up
```

To stop:

```
docker compose down
```

Monitoring:

```
docker compose logs
docker compose ps
```

What is the beauty of our docker compose? First, we can download the whole repository and install it in the client's computer without sweating too much...

Moreover, we can also easily scale our service, for example by running 2 workers instead of one!

```
docker-compose up --scale worker=2
```

There are other improvements that can be easily implemented with this architecture. Can you figure out them?

There are other improvements that can be easily implemented with this architecture. Can you figure out them?

- scale web container and add load balancer

There are other improvements that can be easily implemented with this architecture. Can you figure out them?

- scale web container and add load balancer
- caching machine learning results