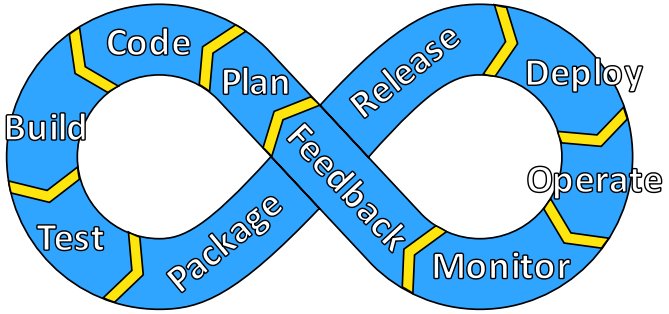



# DevOps & Cloud - 2

Containers & Micro services

# DevOps – Containers & Micro services

- **Last time** was a general introduction and overview
  - Technologies, tools
  - Using Git etc.
- **This time** we'll get our hands dirty on specific tools, features & code
  - Containers
  - Micro services
- This a LOT – so we'll skimp on some details, but think of it as a "map" of how to get going with containers & microservices
- With the map, you can fill out the details yourself (Google's your friend)

# Overview

1) **Containers – what, why and how**
2) Dockerfile's – creating our own images/containers
3) Docker compose – managing multiple containers
4) Development containers (if time permits)
5) Microservices – what, why and one way to do it
6) Exercise: working with containers and a microservice

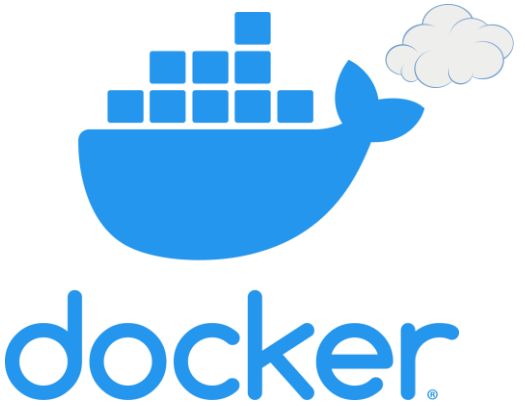
# Containers – what we need

- To get up and running w. containers on Windows we need to install:
  - Windows Subsystem for Linux v2 (must be installed first)
  - Docker Desktop
  - Windows Terminal
- In addition Visual Studio Code (VSCode) works REALLY well w. containers, so it's a good idea to install that as well

# Installing WSL

- To Install WSL (II) Click onto:
- https://learn.microsoft.com/en-us/windows/wsl/install & https://learn.microsoft.com/en-us/windows/wsl/setup/environment
- Let's follow the steps together ….
  - Install WSL itself
  - Setup username/password
  - Do initial update (`sudo apt update;sudo apt upgrade`)
  - Install packages you need, example: `sudo apt install git`
    *Note: Don't install Docker inside Linux!*
  - Install and setup Windows Terminal
    - Default profile, colors, font etc ……..
  - Add Windows Terminal to the process line for easy access

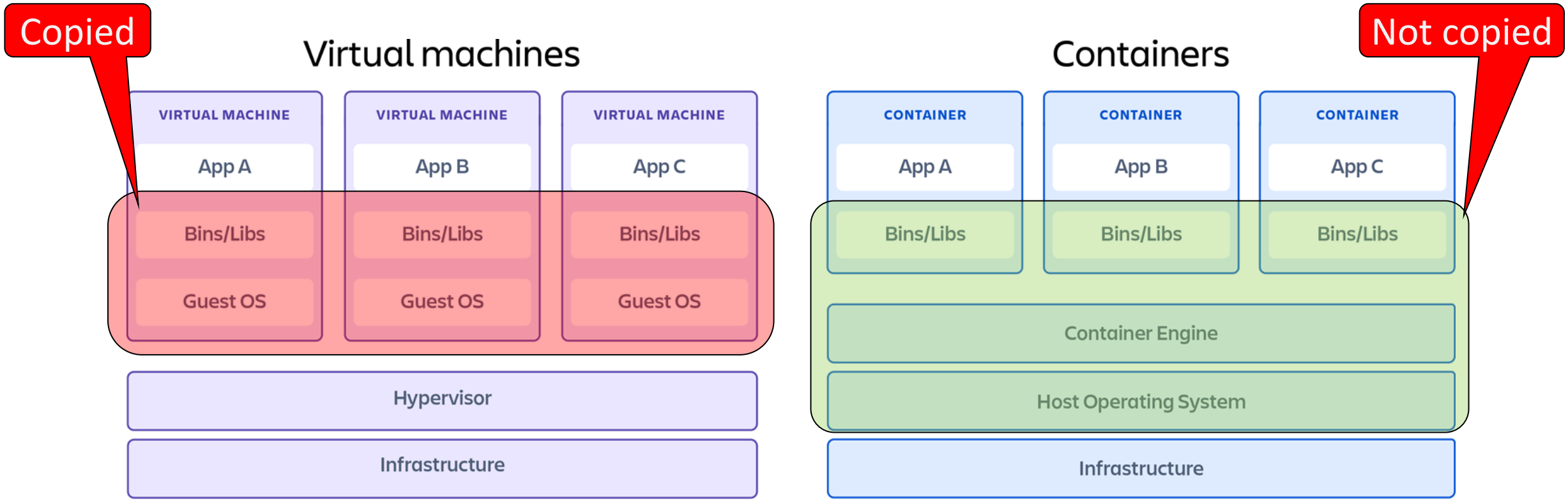
# First dip in the Docker sea




- Docker is the company that popularized Linux containers
- A container is a kind of “virtual machine”, in the sense that apps running inside a container sees their own “private” machine
  - They have their own file system, nobody from outside the container can mess with files
  - No other processes are running
  - They see their own network (devices and network addresses)
  - All needed libraries have the exact right, well-defined version
  - An app may run with root/superuser rights
- Seen from the out side the container is a fully closed and sandboxed jail
  - Apps running inside the container cannot see or modify files, processes or access the network outside the box

# Containers vs tradition VM's

- Containers start & stop real quick since they don't have to simulate a full machine w. hw and everything, VM's have to do a full boot/shutdown cycle
- Real VM's duplicate all stuff used across different VM's, containers don't

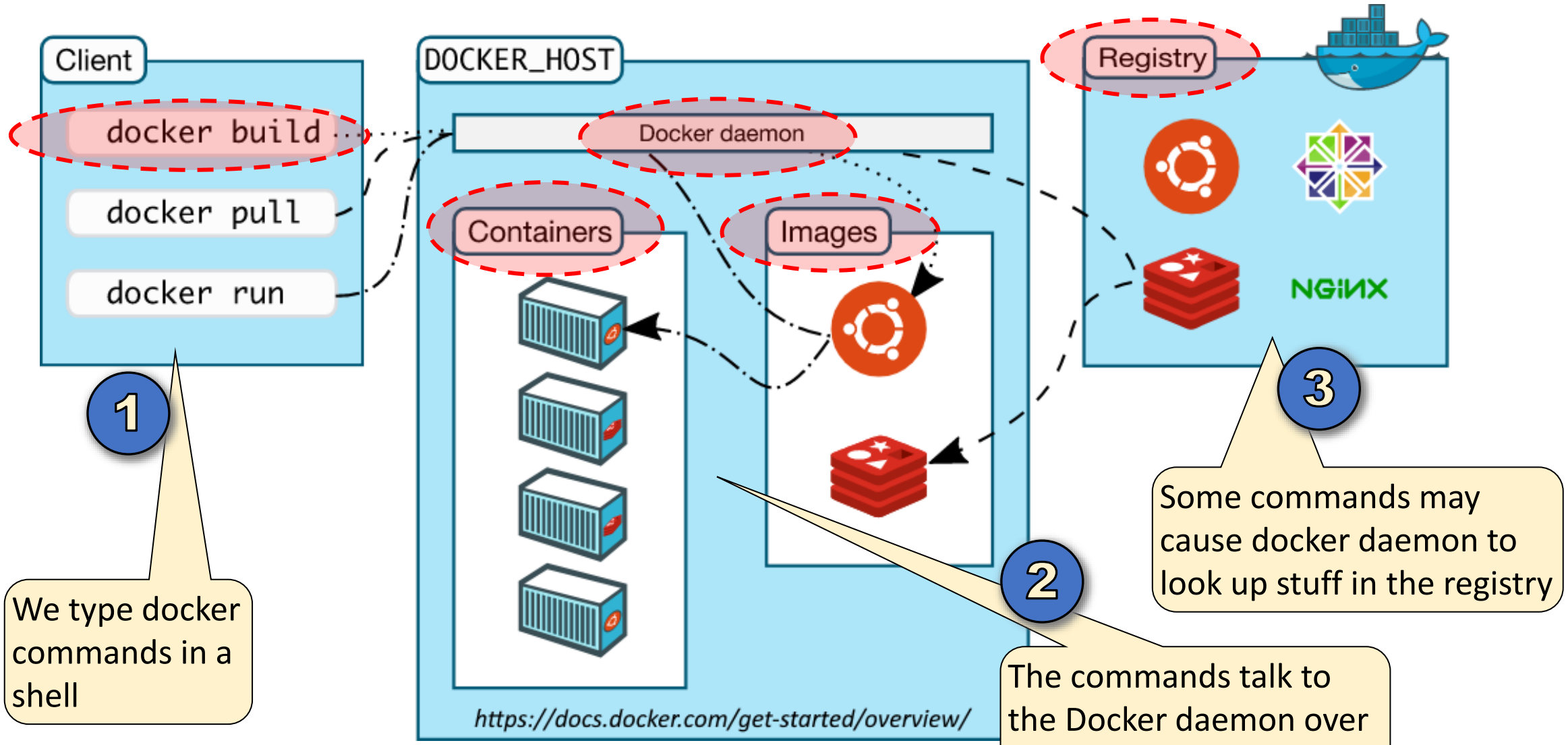



https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

# Docker schmocker, Podman and containers

- Docker (the company) popularized containers, and we often use the two, Docker and containers, as if they are one and the same
- They are not – there are other providers of container technology on Linux, such as Redhat Linux that makes Podman
- So, informally Docker and Podman are two examples of *container management "engines"* allowing us to build, run and manage containers
- Container tech is standardized to an actually useful level, e.g. via the OCI (Open Container Initiative)
- We'll only work with Docker containers, but what we learn by working w. Docker works very well to other container types (e.g Podman)
- But henceforth we'll only talk about Docker containers

# How Docker works



# Docker images, containers and such …

- ***Client:*** An application that wants to do Docker stuff. Example: A shell running in a terminal, where we manually execute Docker commands.
- ***Dockerd:*** The actual container management engine that runs as a service which the clients can talk to over a (usually local) network connection (i.e. client and docker host is often the same machine)
- ***Registry:*** One or more registries ("library") of premade docker images ready to be downloaded and used
- ***Image:*** A read-only template with instructions for creating a Docker container, often based on another image with some additional customization.
- ***Container:*** A runnable instance of an image.
- ***Volume:*** A persistent "file system" that can be attached to a container.

# Docker Hub – Docker's own registry

- Not just a nice collection of images that we can pull, but also a place with important info about the images
- Lets have a look-see …. https://hub.docker.com/
- Using the right images
  - Features vs size
  - Versions/tags (important!) – in general don't use :latest but use a fixed version and then do a deliberate and considered upgrade when it's needed
- There are other registries, one of the more interesting is Github Container Registry (ghcr.io) which is easy to use integrate into automated GitHub actions (it really should be called Github *Image* registry)

# Containers in a nutshell

- Think of a ***container*** as a lightweight virtual machine
- ***Images*** are the software we run inside the containers
  - The same image can run om multiple containers at the same time
  - Once we have created the image we can run as many copies as we like
- A **registry** is a website where we
  - Go to fetch images, or
  - Go for uploading and sharing images

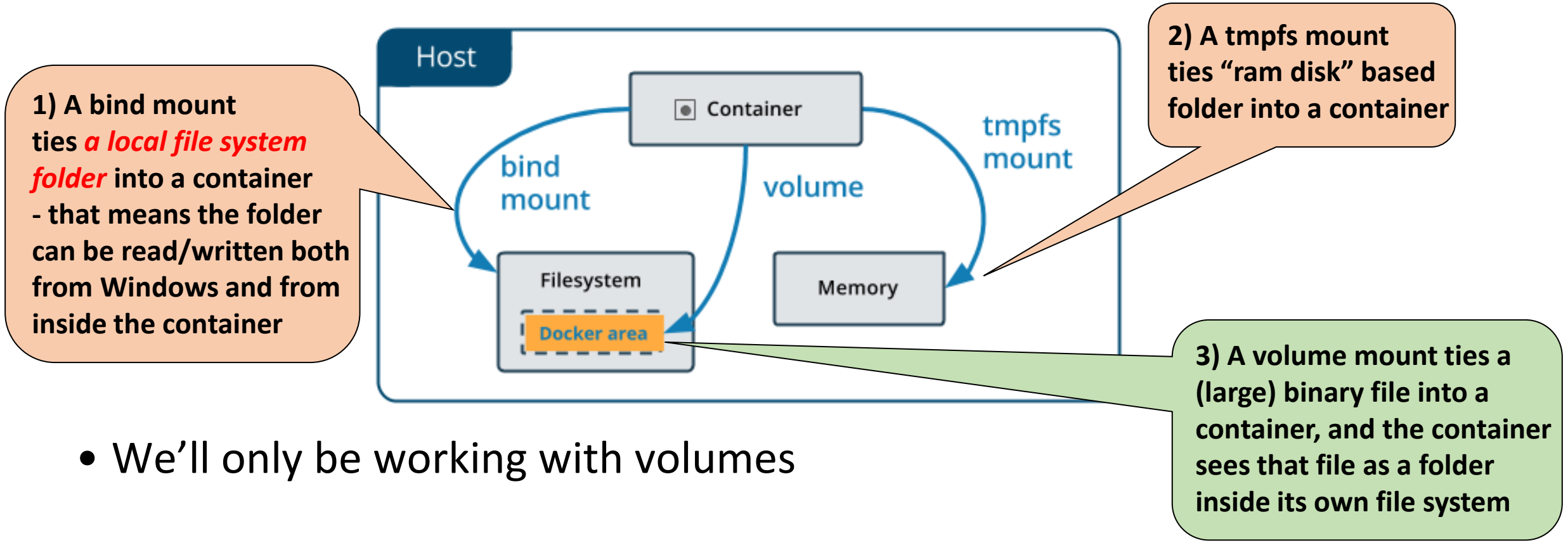
# Basic docker operations

- All basic docker operations can be done via commands
- There's excellent documentation, lets take a quick look at that: https://docs.docker.com/reference/
- Let's do a rundown of the most basic commands with the most commonly used options
  - ***Containers:*** create, start, run, stop, list, delete
  - ***Images:*** pull, list, delete, build (for next session)
  - ***Volumes:*** (see next slide) create, list, delete


- Some commands are "macros" that actually translates into multiple basic commands, such as *run = (pull if needed)+create+start*

# Docker volumes

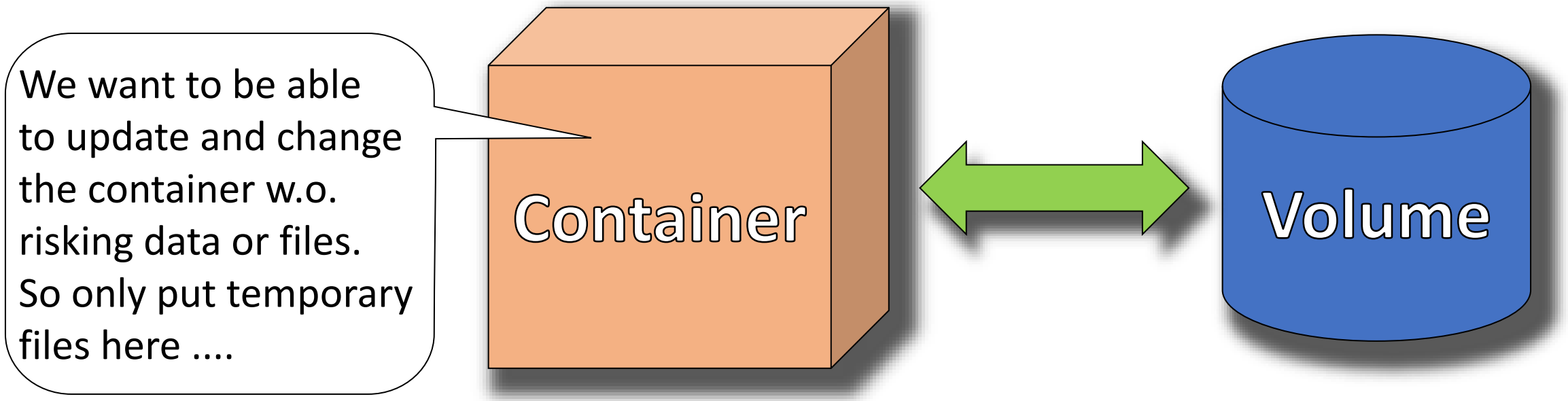
- There are 3 ways we can provide “disk space/a file system” for our containers




- We’ll only be working with volumes

# Docker volumes – a smart thing ☺

- Containers should be create-use-throw away things – i.e. don't accumulate valuable files/data inside your containers
- Use volumes for valuable stuff
- Volumes can be backed up, shared across containers etc.



# Pulling and running a Docker image

1. Follow me .... but first let's take a tour of the Docker Desktop GUI
2. Let's look in Docker Desktop .... no hello-world image/container there

   ```
   > docker pull hello-world
   ```

3. Now let's look in Docker Desktop again .... there's an image!

   ```
   > docker run hello-world
   ```

4. Now let's look in Docker Desktop again .... now there's both an image and a container

   ```
   > docker start -ai crazy_golick
   ```

5. We can run the container as many times we like
6. We can delete the container and image if we like .....

Replace with your own random container name

# Exercise 2-1

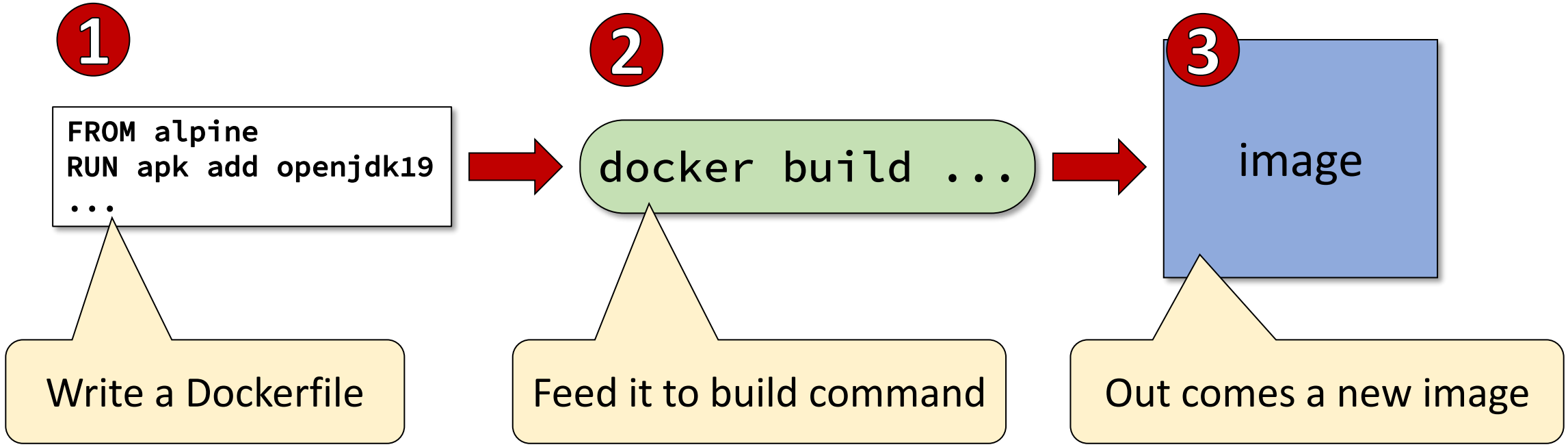
- Pulling and running your first image
- 20 minutes

# Overview

1) Containers – what, why and how
2) **Dockerfile's – creating our own images/containers**
3) Docker compose – managing multiple containers
4) Development containers (if time permits)
5) Microservices – what, why and one way to do it
6) Exercise: working with containers and a microservice

# Creating our own images w. customized features

- To create our own Docker images, we need to look at 2 (closely) related things
  - The `docker build` command
  - Writing a `Dockerfile`



# The Dockerfile

- A Dockerfile is a simple text-file, normally called …. Dockerfile
- Here's an example:

*Comments*

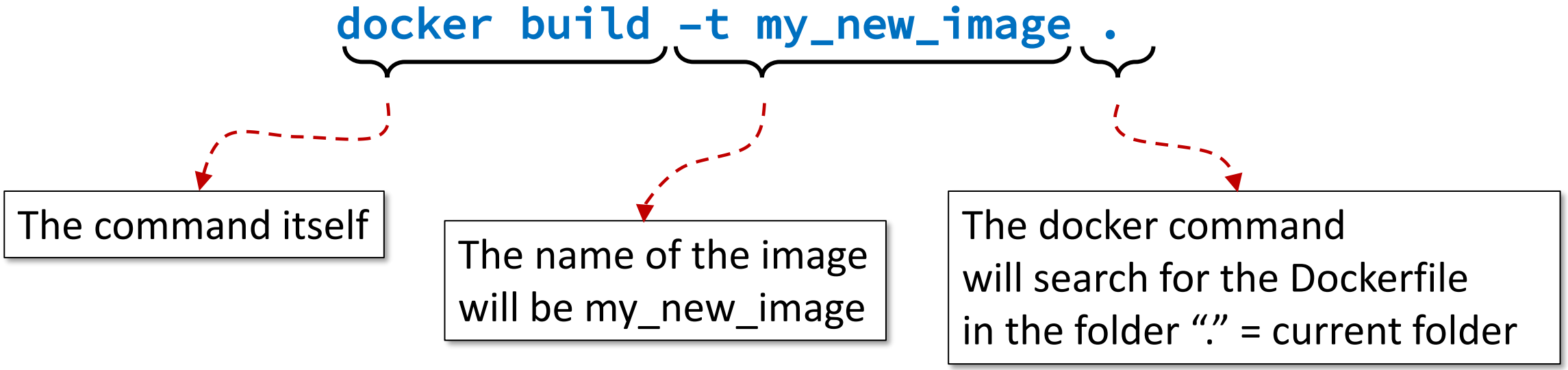
*Instruction*

```
# This is comment
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn -production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

So a Dockerfile is basically a list of instructions, perhaps with some comments

# The docker build command

- To use a Dockerfile to create an image we use the docker build command
- Example:



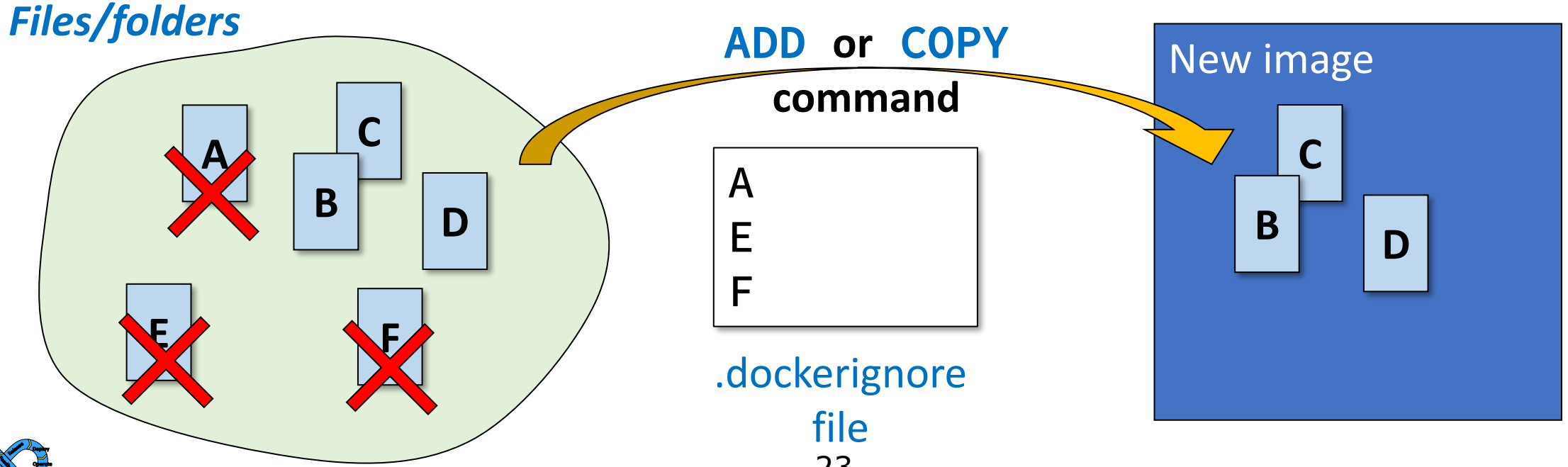

- As usual the Docker "manual" is excellent https://docs.docker.com/engine/reference/commandline/build

# The most important Dockerfile instructions

- https://docs.docker.com/engine/reference/builder/
- The essentials
  - **FROM** – specify base image
  - **WORKDIR** – specify directory for *RUN*, *CMD*, *ENTRYPOINT*, *COPY* and *ADD*
  - **CMD** and/or **ENTRYPOINT** – at least one of these should be present to specifyythe command to run by default in container
- Almost always used
  - **RUN** – run a command inside the new image that is under construction
  - **ADD** and **COPY** – add files/folders (COPY) or more advanced stuff like files, directories or remote file URLs, Git repos (ADD)
    **EXPOSE** – list ports needed (for apps that wants to accept network connections)

# Using a .dockerignore file

- If you create a file called .dockerignore in the folder where your Dockerfile is located, it kind of acts like an .gitignore file
- All files that match a pattern in the .dockerignore file will will be ignored in COPY and ADD instructions



# How to do A LOT with a simple Dockerfile

- We'll try this Dockerfile, from ***appl_container_demo_simple.zip*** :

```
FROM nginx:1.23-alpine-slim
COPY static-html-directory /usr/share/nginx/html
ENV NGINX_ENTRYPOINT_QUIET_LOGS=1
ENV NGINX_HOST=doc.doc
ENV NGINX_PORT=80
```

- Download the example to your WSL and unzip it
- Let's look at the structure/contents, then do this in a shell:

```
cd appl_container_demo_simple
docker build -t nginx_docweb .
docker run -p 8888:80 --rm nginx_docweb
```

- Start browser at ***localhost:8888***

# Here's another Dockerfile example

- If doing Java development, you need to create a runnable jar-file (like Maven does when you execute the package goal). Once you have the ja-file:
- Then all you need is in the Dockerfile is:

```
FROM eclipse-temurin:17
WORKDIR /tmp
COPY ./target/MyApplication-1.0-SNAPSHOT.jar .
ENTRYPOINT ["java","Main"]
```

The Java version you need

The path to the executable jar-file

# Exercise 2-2

- Create and run your first image
- 30 minutes

# Overview

1) Containers – what, why and how
2) Dockerfile's – creating our own images/containers
3) **Docker compose – managing multiple containers**
4) Development containers (if time permits)
5) Microservices – what, why and one way to do it
6) Exercise: working with containers and a microservice

# From one container to many containers

- If we start doing something like microservices, or simply create applications which naturally are split in multiple “runnables” (e.g. a backend with accompanying database), we end up with multiple executables
- If we really want to decouple each executable as much as possible, from each other, from the platform, from specific technologies and deployment wise, we should put each executable in its own container
- We have seen that containers are smart for wrapping a single executable, but what happens if we need to start multiple executables (and hence containers) to run our application?

# Handling multiple containers

- Since containers have grown so popular so fast, it's no wonder that tools for managing multi-container applications already exist
- For Docker containers we can use Docker Compose and for general, heavy duty, cross-domain/cross-machine container orchestration we can use Kubernetes (aka. K8s)
- For doing management for a single host machine, Docker Compose is fine, and that's what we'll be looking at here
- K8s is needed if we want to handle stuff that spans multiple machines/domains. On top of that K8s supports:
  - Scaling, to automatically create more instances at runtime to meet increased demand.
  - Fault-tolerance, to replenish failing pods (Docker Compose can only restart)

# The docker compose file

- We know about Docker files, know we need to learn about Docker composer files
- Once we have create a suitable docker compose file, all we need to do is type (to start, respectively stop our application)
  - `docker compose up`
  - `docker compose down`
- Docker compose will then
  - Build the necessary images (if needed)
  - Create and start containers
  - Connect our containers together on the network
  - Monitor and restart failing containers

# A docker compose file – docker-compose.yml

```yaml
version: "3.8"
services:
  db:
    image: "postgres:15.2-alpine"
    restart: always
    container_name: "pgsql-container"
    ports:
      - 5432:5432
    volumes:
      - db:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=microsvc-db
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
  backend:
    image: 'microsvc-demo:latest'
    build:
      context: ./
    container_name: 'microsvc-demo-container'
    ports:
      - 8080:8080
    depends_on:
      - db
    environment:
      - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/microsvc-db
      - SPRING_DATASOURCE_USERNAME=postgres
      - SPRING_DATASOURCE_PASSWORD=postgres
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
volumes:
  db:
    driver: local
```

# A complete example

- We'll study this example: https://github.com/DaFessor/devops-microsvc-demo
- Multiple containers
- Uses Docker compose
- Actually an example of one way to implement a microservice

# Exercise 2-3

- Use Docker compose to start and stop multiple containers
- 30 minutes

# Overview

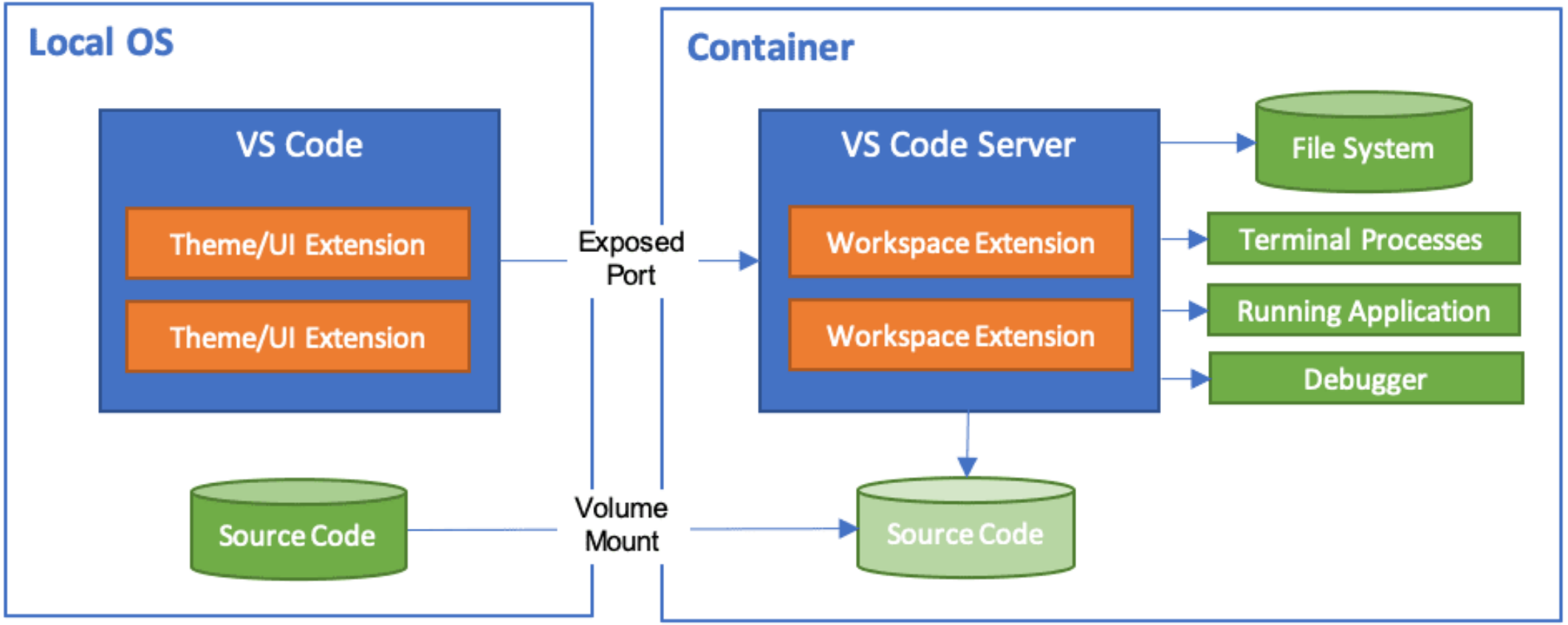
1) Containers – what, why and how
2) Dockerfile's – creating our own images/containers
3) Docker compose – managing multiple containers
4) **Development containers (if time permits)**
5) Microservices – what, why and one way to do it
6) Exercise: working with containers and a microservice

# VSCode + containers =  BFF

- VSCode has very good support for working with containers
- We can use VSCode for 2 different things with respect for containers
  - We can “jump onto” any container and then view, modify and create files
  - We can open any Docker volume or Git-based project inside a *development container*

# VSCode development containers

With VSCode we can run our entire development environment/toolchain inside a container:




https://code.visualstudio.com/docs/devcontainers/containers

# Development containers

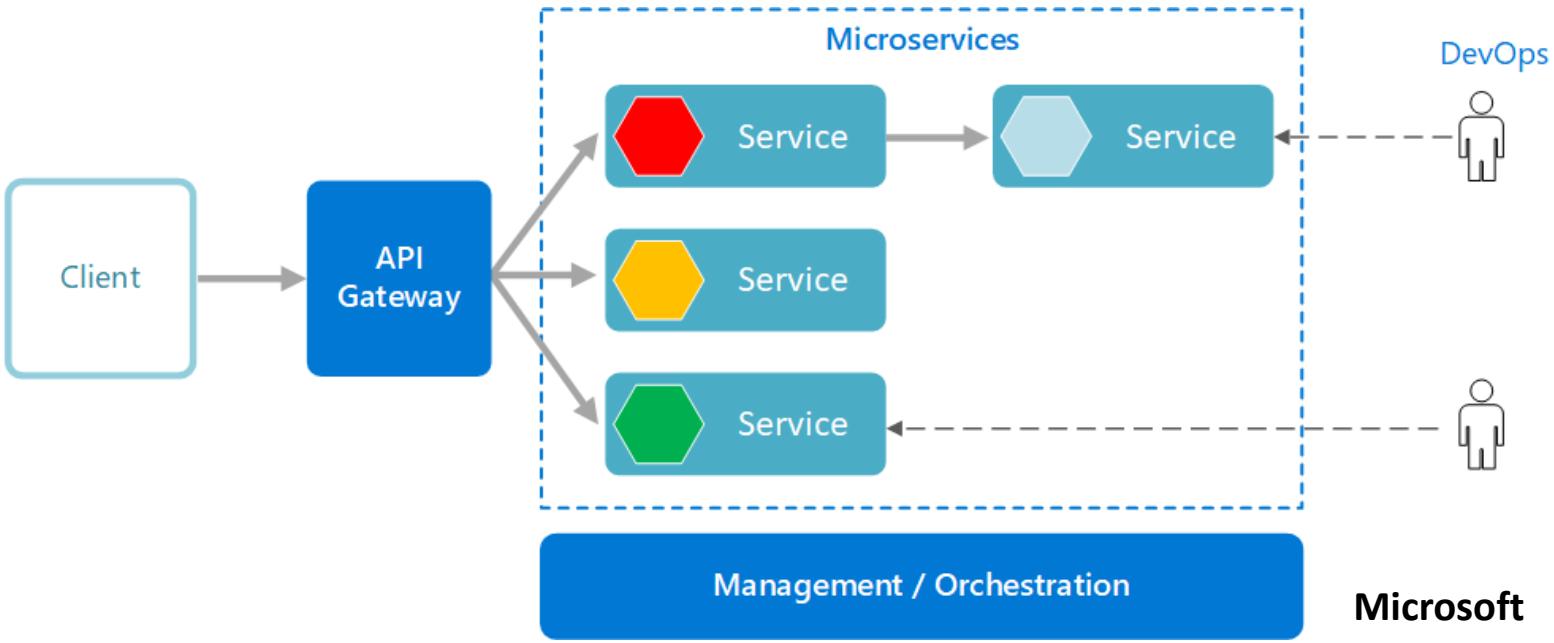
- Why would it be smart to have your entire development environment running in a container?
- For several reasons:
  - Configuration as code – we can control versions of all parts of our build tools, since the config files that control our build tools can be placed under Git control
  - We can use the exact same container/image to build our code locally and in our GitHub/GitLab pipeline
  - We can know for sure that all team member use the exact same version of the build tools/chain (version of Java, Maven, libraries, etc.)
- Demo: https://github.com/DaFessor/iot-base – complete IoT/embedded development environment in a container

# Overview

1) Containers – what, why and how
2) Dockerfile's – creating our own images/containers
3) Docker compose – managing multiple containers
4) Development containers (if time permits)
5) **Microservices – what, why and one way to do it**
6) Exercise: working with containers and a microservice

# Microservices – the top level view

- An architectural style, where we divide our application into a collection of small autonomous services
- Each service is self-contained, independently developed and maintained, and is only concerned with a single functionality/business capability



# Microservices – what are they? (1)

Microsoft

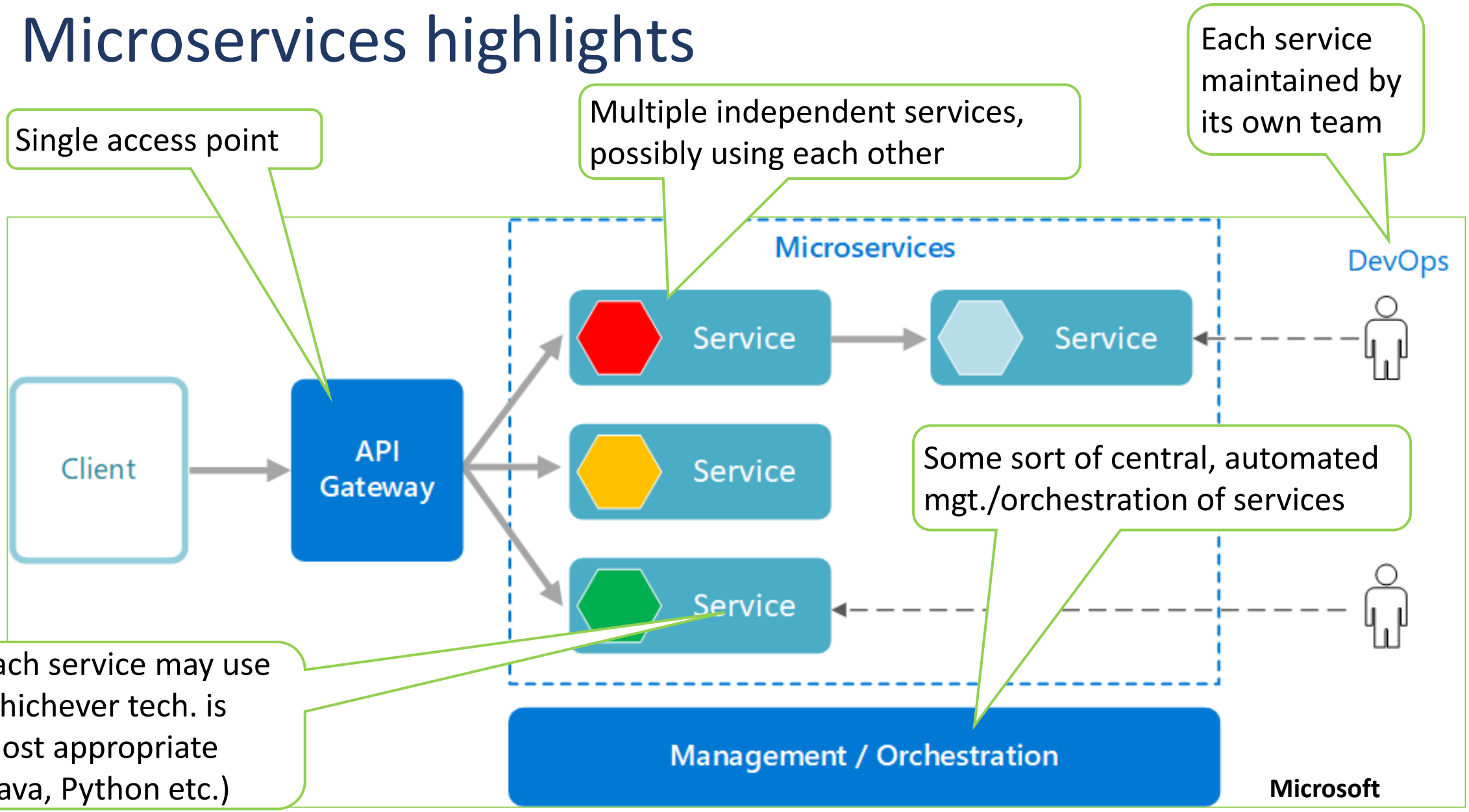
1. Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
2. Each service is a separate codebase, which can be managed by a small development team.
3. Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
4. Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.

# Microservices – what are they? (2)

Microsoft

5. Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
6. Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.

# Microservices highlights



# Overview

1) Containers – what, why and how
2) Dockerfile's – creating our own images/containers
3) Docker compose – managing multiple containers
4) Development containers (if time permits)
5) Microservices – what, why and one way to do it
6) **Exercise: working with containers and a microservice**

# Exercise 2-4

- Containerize a small mini-app that uses a db (sort of a clone of the https://github.com/DaFessor/devops-microsvc-demo example, but you have to build it from scratch)