



Aalto University
School of Electrical
Engineering

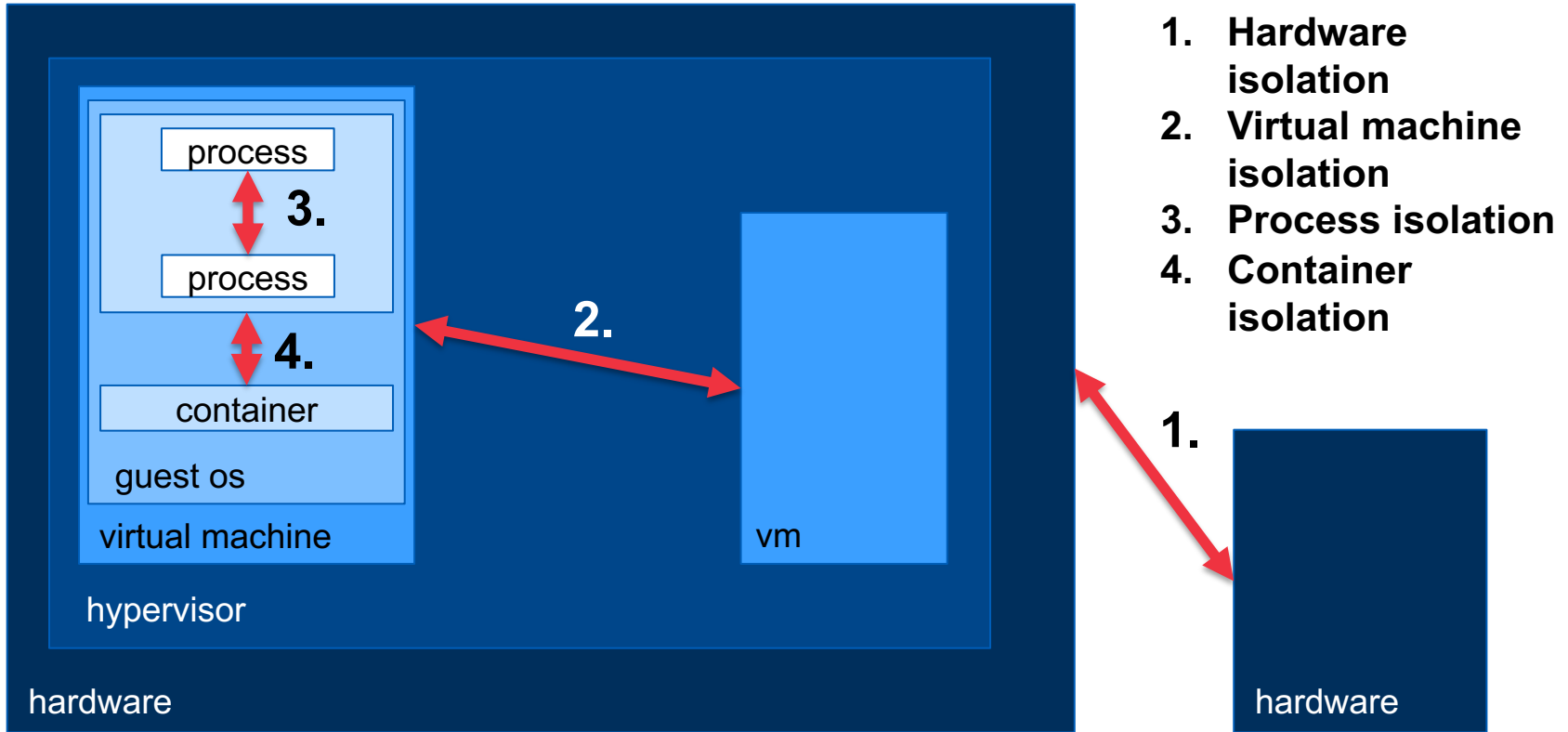
Containers: Docker and Kubernetes

17.1.2019

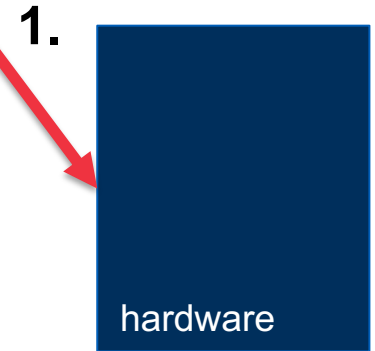
Santeri Paavolainen

“**Operating-system-level virtualization**, also known as **containerization**, refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances. Such instances, called **containers**, partitions, virtual environments (VEs) or jails (FreeBSD jail or chroot jail), may look like real computers from the point of view of programs running in them. A computer program running on an ordinary operating system can see all resources (connected devices, files and folders, network shares, CPU power, quantifiable hardware capabilities) of that computer. However, programs running inside a container can only see the container's contents and devices assigned to the container.”

Wikipedia: [Operating-system-level virtualization](#)

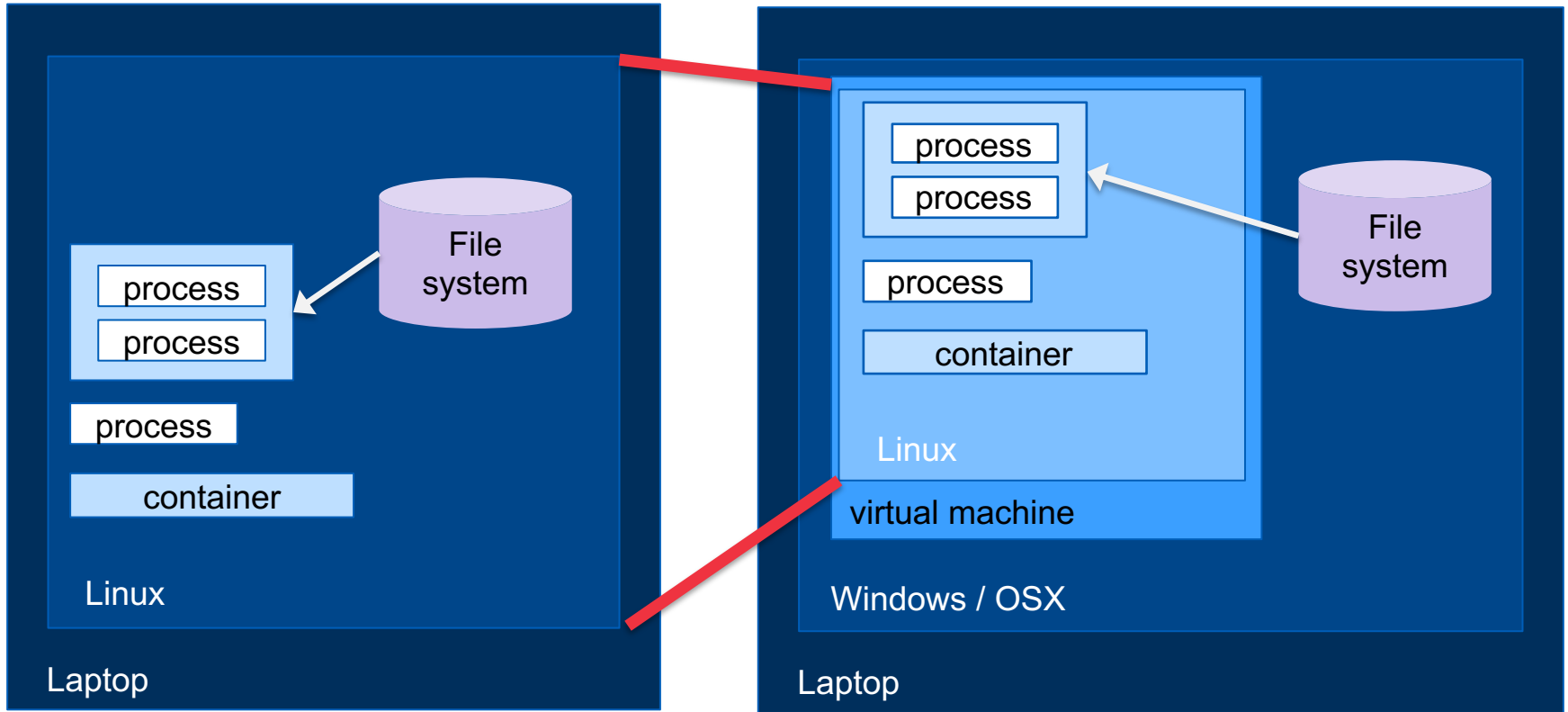


1. Hardware isolation
2. Virtual machine isolation
3. Process isolation
4. Container isolation



Container technologies

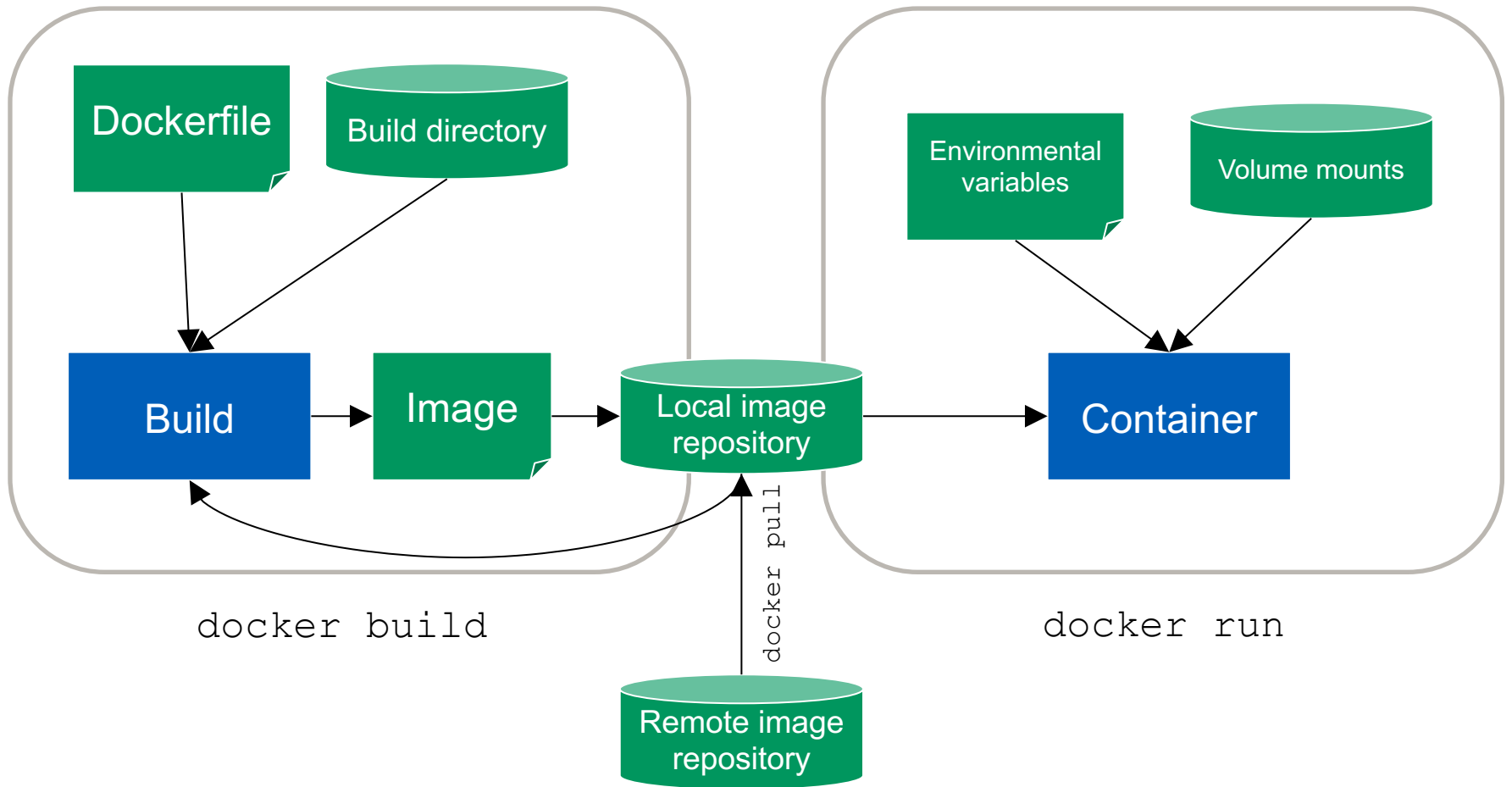
- **LXC: Linux Containers**
 - OS-level process isolation = cgroups + isolated namespaces
- **Docker uses LXC**
- **Windows ...**
 - Windows Server Containers and Hyper-V Isolation, **but ...**
- **Reality: Windows and MacOS**
 - Docker actually runs a Linux VM where Docker uses LXC
 - The fact that containers are run in a separate virtual machine can cause issues with volume mounts and networks!
- **There is also docker-machine**
 - Which is different from Docker for Windows and Docker for Mac
 - Useful if you want to run a “container swarm” in different VMs locally



Why use Docker for microservices?

- **Docker creates containers from images**
 - Images themselves are immutable → identical versions in multiple environments
 - Image repositories store images (public, private & local);
Location of a single image is a registry, images are tagged (often for versioning)
- **Images themselves built ... using containers**
 - Building images itself isolated from host computer* → “Dockerfile” build script cannot escape into host computer!
- **Containers are isolated from outside unless explicitly exposed**
 - Network ports and file system mount points
- **Containers also isolated from other containers unless share a virtual network**
 - Note that by default, they do share a common network

* mostly



[version.aalto.fi/gitlab/
microservices-serverless-course/
course-samples](https://version.aalto.fi/gitlab/microservices-serverless-course/course-samples)



Aalto University
School of Electrical
Engineering

Simple example

- **Shows basic docker commands**
 - `docker ps`
 - `docker build`
 - `docker run`
 - `docker rm`
 - `docker images`
 - `docker rmi`
- **Dockerfile**
 - FROM
 - CMD
- **Goal: Print “Hello, world!” on screen**

Long-running commands

- **What happens if there is a long-running process (server)**
- **Some more useful commands for debugging**
 - `docker ps`
 - `docker exec`
 - `docker stats`

Modifying image

- **How to modify the image?**
- **Dockerfile**
 - RUN
- **Let's install bash**
 - Note: alpine includes /bin/sh, we could have used that in previous example already (`docker exec ... /bin/sh`)

Simple web static web server

- **How do we get files into the image?**
- **Dockerfile**
 - COPY
 - WORKDIR
- **Plus, how do we access HTTP on the container**

A bit more on docker networking

- **We should define what ports the container exposes in the Dockerfile**
 - `EXPOSE 80`
 - `EXPOSE 80/tcp`
- **This does not automatically publish them**
 - "Publishing" means allowing access from outside the container network
 - `docker network ls`
 - `docker run --network <network>`

Parameterizing containers

- **Passing arguments to a running container**
 - Command arguments
 - Environment
 - Volume mounts
- **Dockerfile**
 - ENTRYPOINT
 - ENV
 - VOLUME

Parameterizing containers

- **Passing arguments to a running container**
 - ~~Command arguments~~
 - Environment
 - Volume mounts
- **Dockerfile**
 - ~~ENTRYPOINT~~
 - ENV
 - VOLUME

Building and running containers

- **What we've learned so far**
 - Building simple containers with own modifications and files
 - Inspecting running containers (`docker exec`)
 - Exposing and accessing network services in containers (`EXPOSE` and `docker run -p`)
 - Command line arguments (`ENTRYPOINT` and `docker run`)
 - Environmental variables (`ENV` and `docker run -e`)
 - Volume mounts (`VOLUME` and `docker run -v`)

State in containers

- **Previously used `--rm` to not leave containers lying around after they're exited**
- **Persistency with containers is possible by either**
 - Not removing the container – it will retain its local modifications in file system
 - Using a volume that is retained across container lifecycles
- **Retaining state in containers always problematic**
 - However, entirely acceptable for performance reasons (caching) and local development (running database in a retained container)

Example: Local postgres database

```
$ docker run --name mydb -e POSTGRES_PASSWORD=sikret -d postgres
ee7e3301a1bd6a86053ce103f23ccab404a502a89cfe1e3406c89b6f6c61972b
$ docker run --rm -ti --link mydb postgres psql -h mydb -U postgres
Password for user postgres: sikret
psql (11.1 (Debian 11.1-1.pgdg90+1))
Type "help" for help.

postgres=#
```

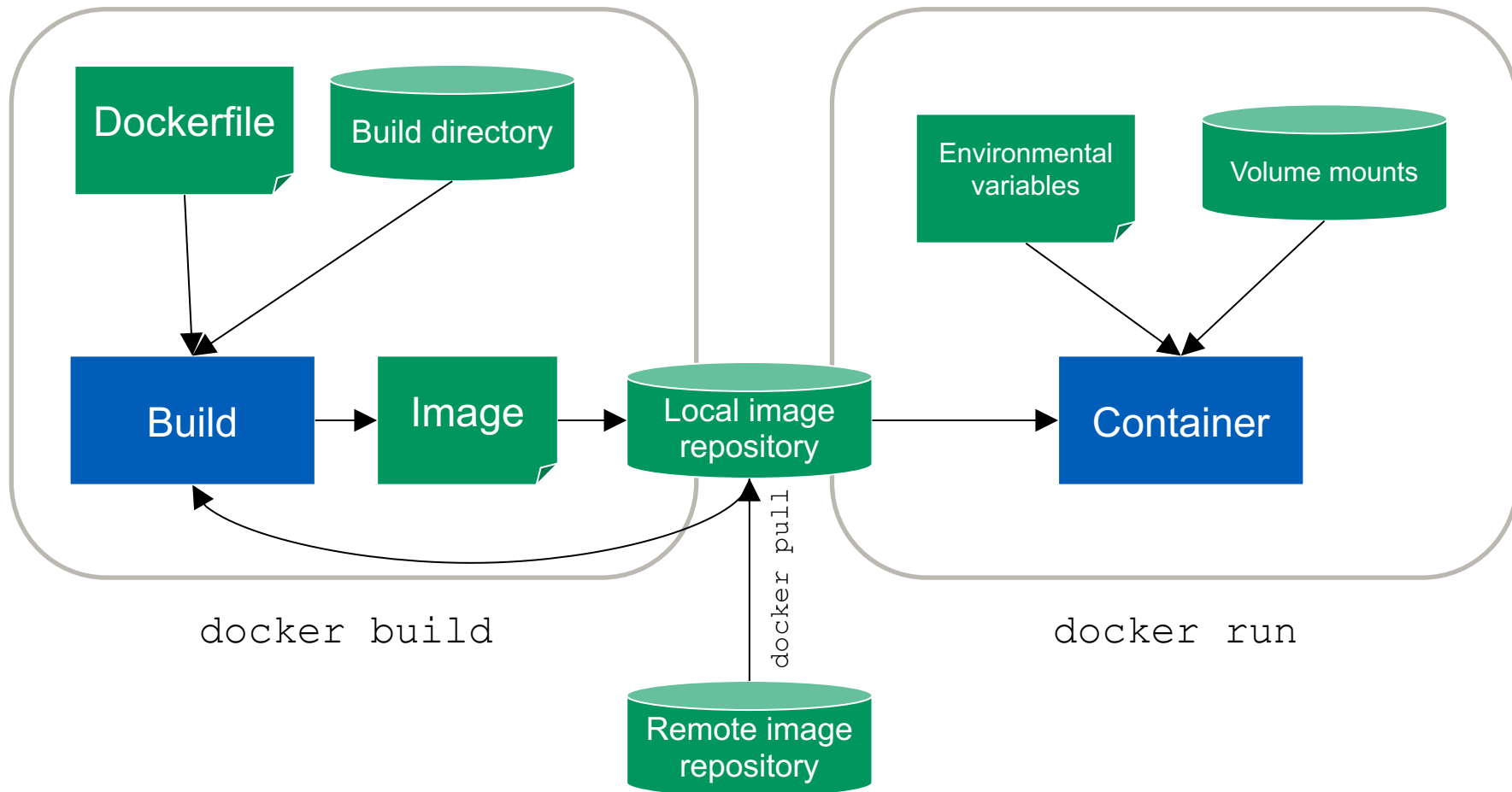

Remote registries

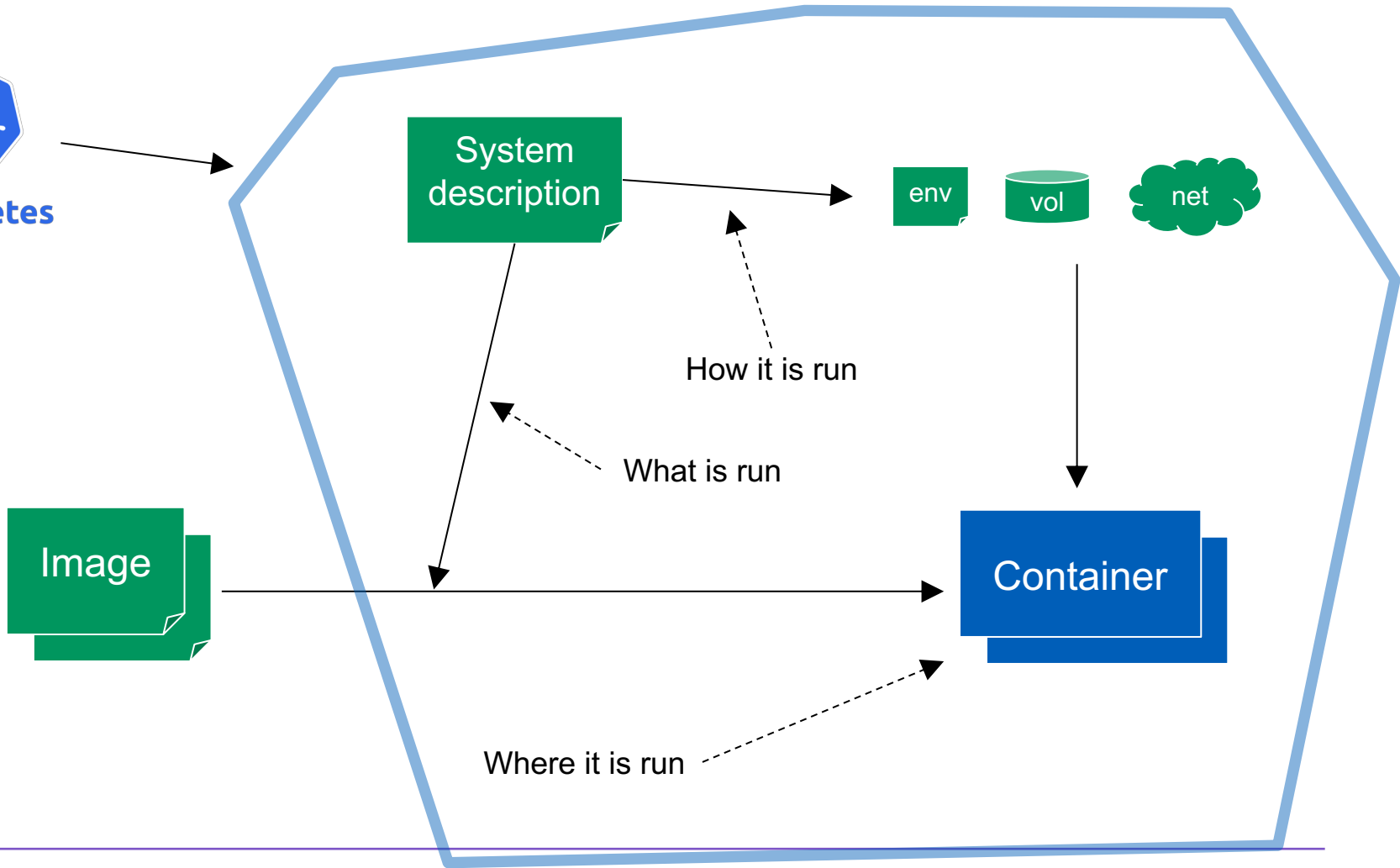
- **Where did “alpine” and “postgres” images come from?**
 - `hub.docker.com/alpine:latest` and `hub.docker.com/postgres:latest`
 - Not URLs!
`<host>/<registry>:<tag>`
- **This is the “docker hub”, centralized & well-known registry location**
 - “hub.docker.com” is implicit for any registry name not found locally
- **You can run your own private registry or registry service**
 - Amazon Elastic Container Registry, Google Container Registry, TreeScale, host your own, ...
 - If developing only locally, not necessary

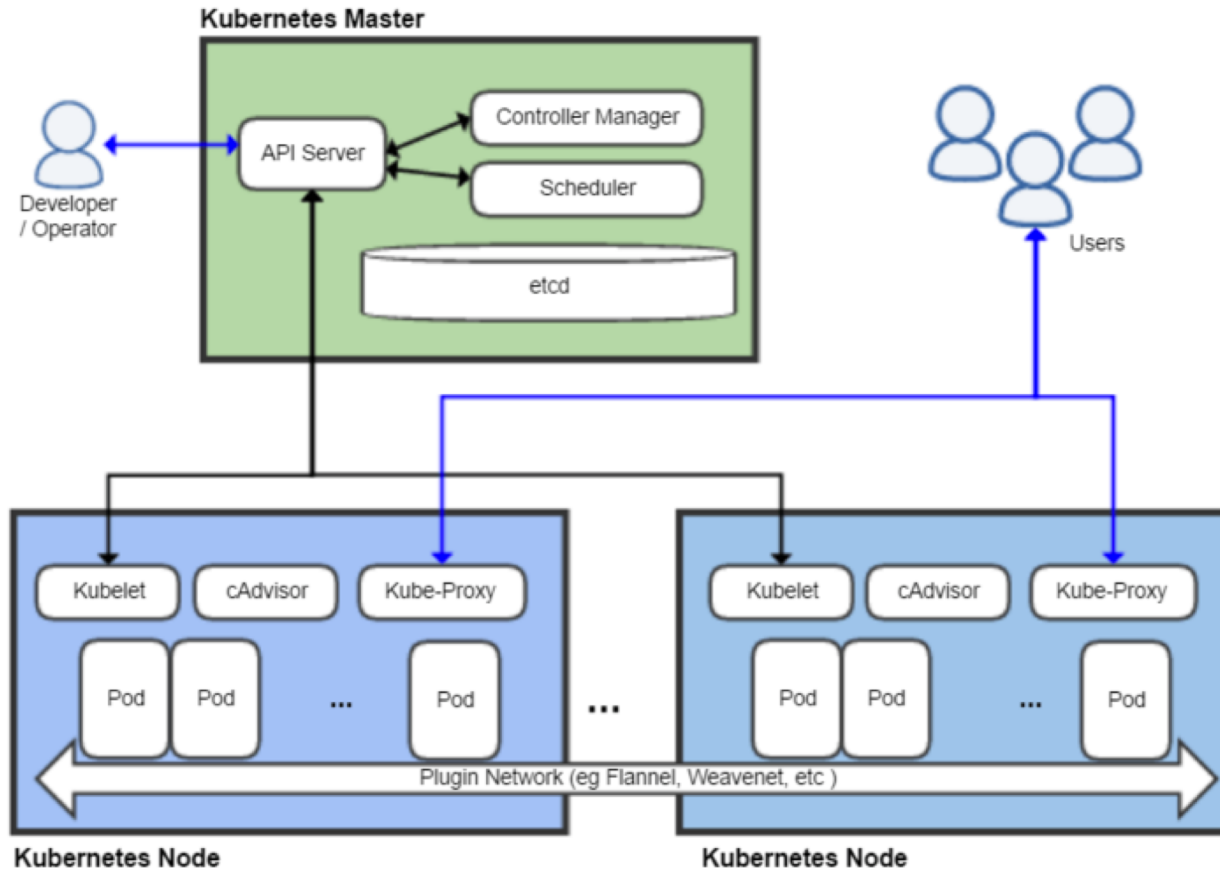
Kubernetes

Container orchestration

- **“docker” itself quite low-level mechanism**
 - To set up a multi-layer service:
 1. *docker network create*
 2. *docker network create*
 3. *docker run x N times*
 4. *Don't forget volumes, and environment, and arguments ...*
 5. *Then remember to start/stop as needed (docker stop, docker start)*
- **”Container orchestration” systems use declarative languages to define what kind of configuration you want to run**
 - Kubernetes, Docker Compose, ...



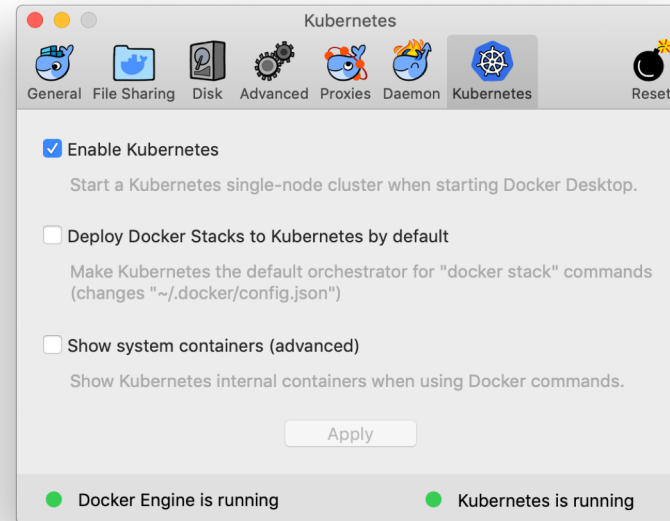




Source: [Khtan66](#) (CC BY 4.0)

Installing kubernetes

- **With UI installers (OSX, Windows)**
 - Already contains kubernetes functionality – but needs to be enabled



Installing Kubernetes

- **Linux (and more advanced for OSX)**
 - Minicube, microk8s (Ubuntu snap)
 - <https://kubernetes.io/docs/tasks/tools/install-kubectl/>
 - Install kubectl and create a local cluster
 - Interactive tutorial: <https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-interactive/>

```
$ kubectl get nodes
NAME                STATUS    ROLES    AGE   VERSION
docker-for-desktop  Ready    master   6m    v1.10.3
```

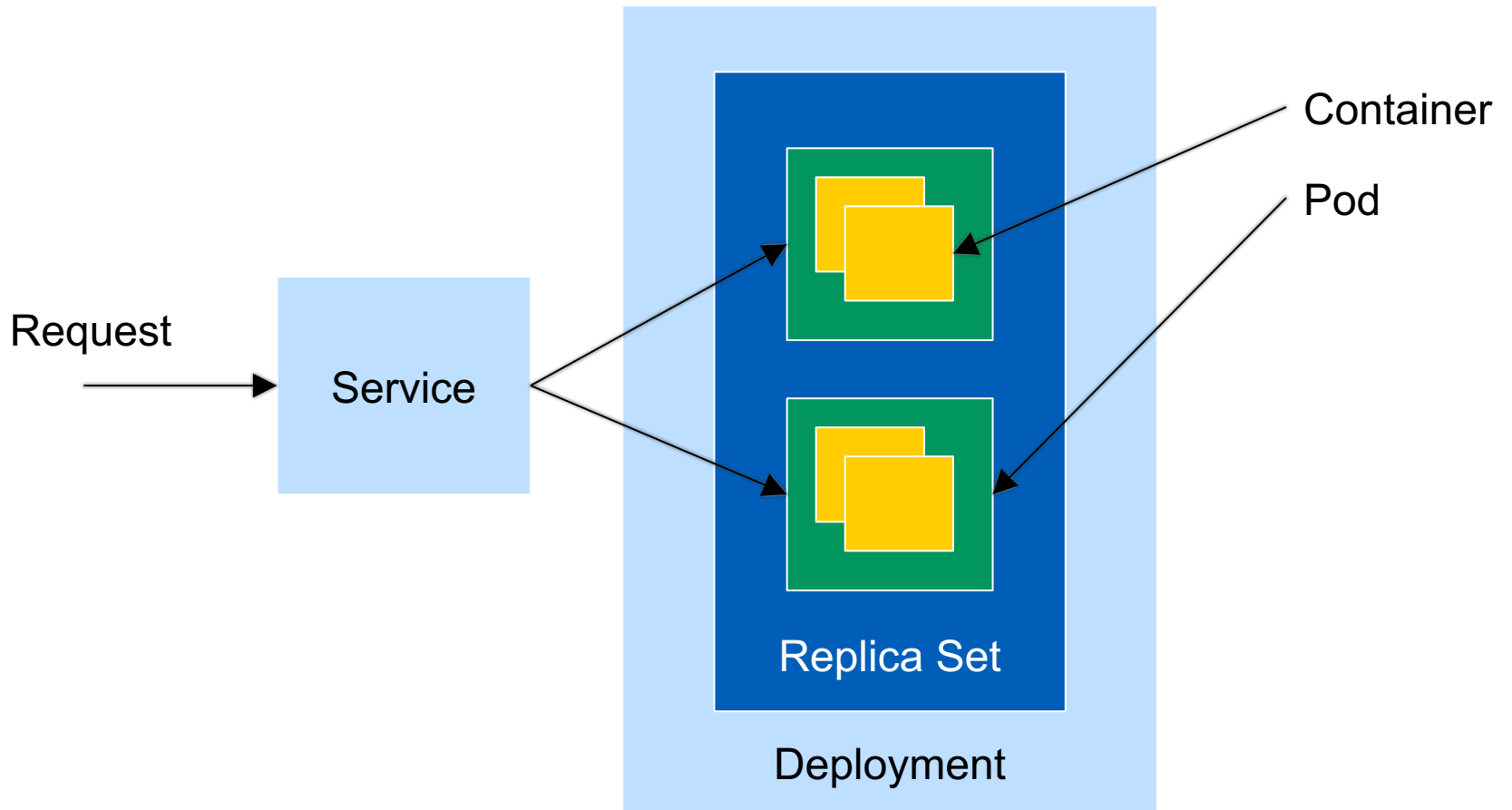
```
$ kubectl get nodes
NAME      STATUS    ROLES    AGE   VERSION
minikube  Ready    <none>   14s   v1.10.0
```


Kubernetes concepts

- **Pods**
 - “A Pod represents a unit of deployment: *a single instance of an application in Kubernetes*, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources.” [[Kubernetes](#)]
 - 1+ containers — tightly coupled, sharing resources, single instance of an application
- **Pods have a limited lifecycle**
 - Controlled by ... a controller
 - Containers within a pod may be restarted without the pod failing

Kubernetes concepts

- **Services**
 - Persistent and long-living
 - Defines how to access (what and how to expose) a specific set of logically identified pods (but does not run pods!)
 - Same pod may be used to provide different types of services
- **Controllers**
 - Responsible for running pods, defined via templates
 - Different pod control models: stateless, stateful, replicated, ...
- **Jobs, Namespaces, Entities, ...**
 - Allow finer control and more elaborate configurations



Registries and Kubernetes

- **This is a practical issue you will run into!**
- **Kubernetes by default tries to pull images**
 - Always if “latest” tag (default) → tries to pull image from Docker Hub which can fail (or fetch image you did not expect)
- **Minikube has internal self-hosted registry**
 - `eval $(minikube docker-env)`
 - Works around the problem ...
- **Docker from docker.com does not (and docker on Linux)**
 - Have to explicitly prevent kubernetes from attempting to pull!
 - `imagePullPolicy: Never`
- **Alternatively use other tag than “latest”**
 - Tries to use local version first, but ... will attempt a pull if not found

Simple example

- Let's run the “hello” container from Docker examples
- Using “Job” controller
 - Useful for one-off operations, batch jobs etc. (not for services)

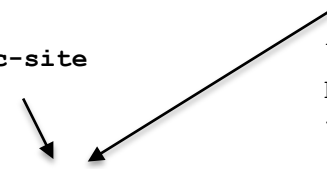
```
apiVersion: batch/v1
kind: Job
metadata:
  name: greeter
spec:
  template:
    spec:
      containers:
      - name: greeter-container
        image: hello
        imagePullPolicy: Never
        restartPolicy: Never
```


Simple web server

- **Re-use “site” from earlier examples**
 - Kubernetes does not address how images are built at all — noticed?
- **Create a “Deployment” type controller**
 - It will internally instantiate a replication set (`replicas` parameter)
 - Note the need of `selector` in both deployment and service
 - *In deployment, it tells what pods the replication set manages*
 - *In service, it tells what pods host the port to expose as a service*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: static-site-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: static-site
  template:
    metadata:
      labels:
        app: static-site
    spec:
      containers:
        - name: site-container
          image: site
          imagePullPolicy: Never
          ports:
            - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  selector:
    app: static-site
  type: LoadBalancer
  ports:
    - port: 80
      name: web
```

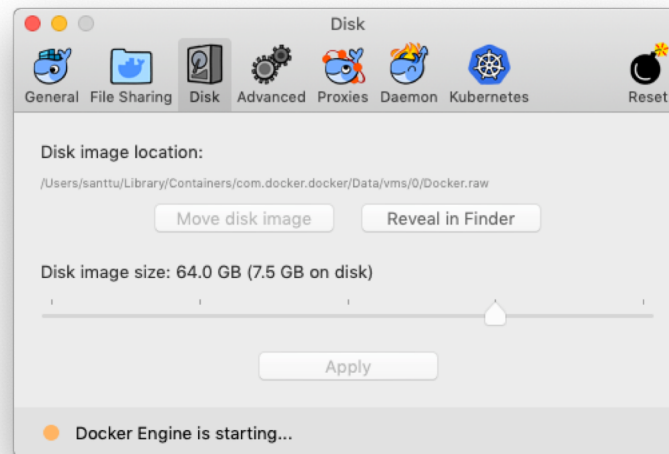


Multiple object in configurations

- **Objects in Kubernetes are persisted**
 - Explicit lifecycle management required: create, delete — these are not scripts!
- **A YAML file can contain multiple sections**
 - first object
 - ----
 - second object
- **Use of service names and namespaces make separation of development, staging and production easier**
 - ```
kind: Service
apiVersion: v1
metadata:
 name: my-service
 namespace: prod
spec:
 type: ExternalName
 externalName: my.database.example.com
```

# What you can and cannot run locally

- **Each container and pod uses memory**
  - Container + everything else (proxies etc.)
  - Java runtime easily >500MB in size
  - Minimal machine-code programs (go, rust) can be replicated (example)
- **Disk space may become an issue too**
  - On OSX/Win local Docker runs in a virtual machine
  - Mostly relevant only if you use a lot of storage
- **Kubernetes objects are persistent**
  - Survives reboots – can't escape replicas=1000 local accident easily
  - Hint: Disable docker desktop autostart ...



# Some topics that will be covered later

- **Covered at appropriate lectures via examples**
  - DNS and service discovery, service proxies, dynamic endpoints
  - Namespaces
  - Different networking modes

# Some practical tips: Writing Dockerfiles

- **Create Dockerfiles in two steps: 1) development and 2) maintenance**
  - In development, just pile up RUN after RUN command
  - In maintenance, optimize images by minimizing RUN commands and layered state, leveraging separate build containers
    - ***Every and each command** in Dockerfile creates a new layer, e.g. difference from earlier state*
    - *“RUN dd if=/dev/zero of=/zeros bs=1G”; ”RUN rm /zeros” defines an image where a single layer contains 1G file*
  - In this course you probably should optimize for speed of development, not minimizing image sizes (aka no to maintenance)

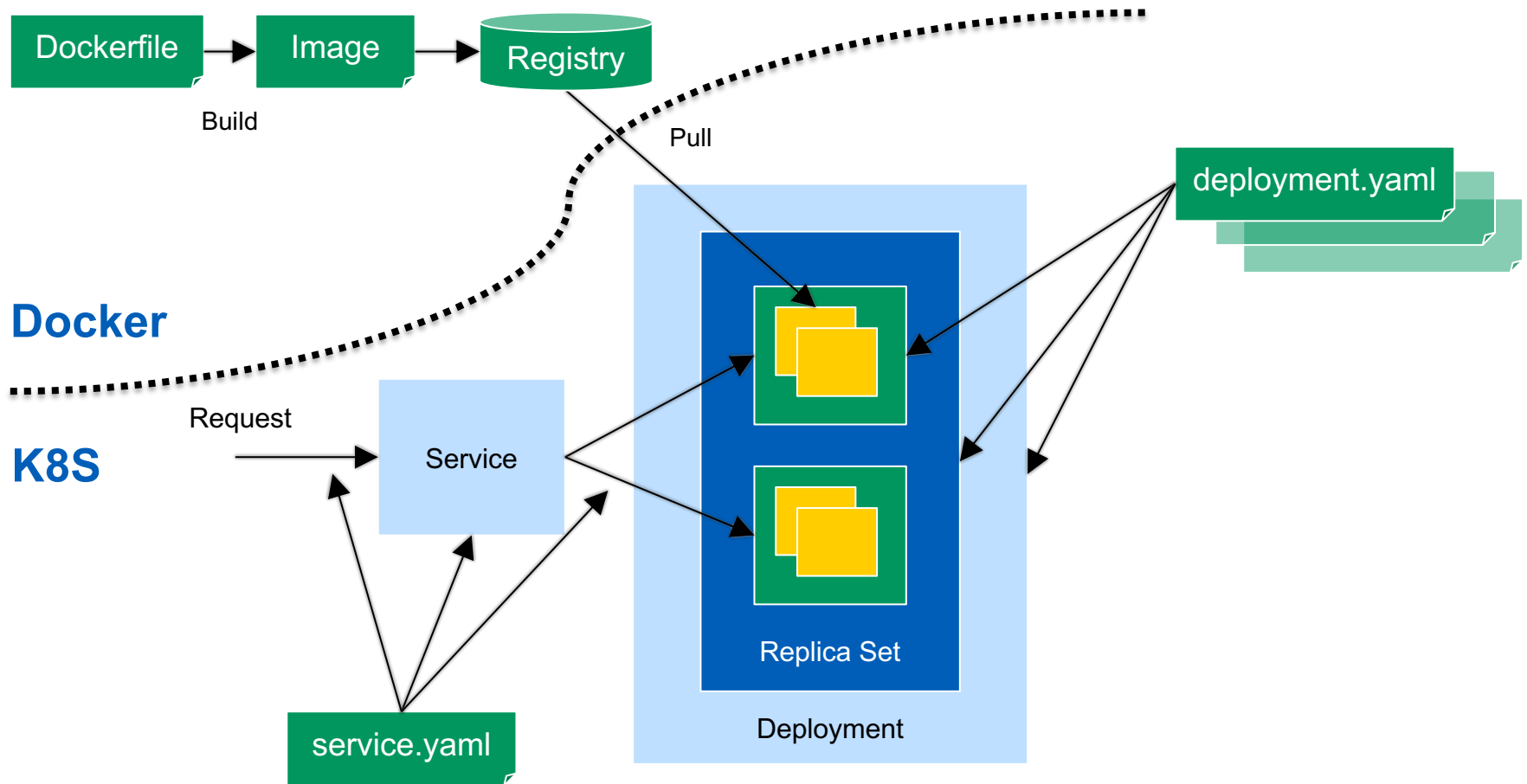
# Writing Dockerfiles

- **Check out Alpine Linux as a small base image**
  - <https://alpinelinux.org/> and [https://hub.docker.com/\\_/alpine](https://hub.docker.com/_/alpine)
  - Package command: apk
  - Install packages: apk add --no-cache <package>
  - Package search: <https://pkgs.alpinelinux.org/packages>
- **Simple workflow**
  - Develop program locally (no docker)
  - Once you get MVP, start to containerize using volume mount
  - docker run --rm -v \$PWD:/work -ti --init alpine
  - cd /work
  - python3 server.py
  - Find out what failed, what was missing, do apk add, go build, sbt, pip install, whatever is needed
  - Add these as RUN to Dockerfile

1

# Summarizing ...







# Summary

- **Docker is a container build and execution framework**
  - Manages networking, volume mounts, registry push/pull, persistent container state, etc.
- **Docker's boundary is a single container**
  - No service orchestration in docker itself (yes in docker compose, but that's a separate solution)
- **Kubernetes widely used for container orchestration**
  - Manages Pods, which can consist of multiple containers, and services which are exposed network ports and/or addresses