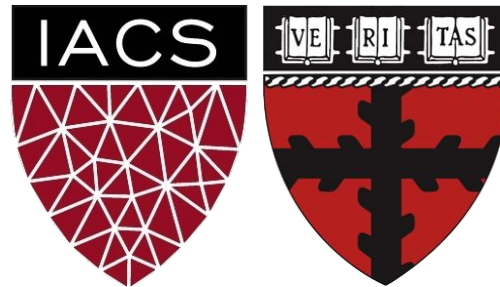


# Containers

CS107/AC207

Pavlos Protopapas

Institute for Applied Computational Science, Harvard



# Outline

---

1. Recap
2. Motivation / Tutorial
3. What is a Container
4. Tutorial: Building & Running Containers using Docker
5. Why use Containers?

# Virtual Environments

---

## Pros

- Reproducible research
- Explicit dependencies
- Improved engineering collaboration

## Cons

- Difficulty setting up your environment
- Not isolation
- Does not always work across different OS

# Virtual Machines

---

## Pros

- Full autonomy
- **Very secure**
- Lower costs
- Used by all Cloud providers for on demand server instances

## Cons

- Uses hardware in local machine
- Not very portable since size of VMs are large
- There is an overhead associated with virtual machines

# Wish List

---

We want a system that:

- Automatically set up (installs) all OS and extra libraries and set up the python environment
- It is isolated
- Uses less resources
- Startups quickly

**Containers**

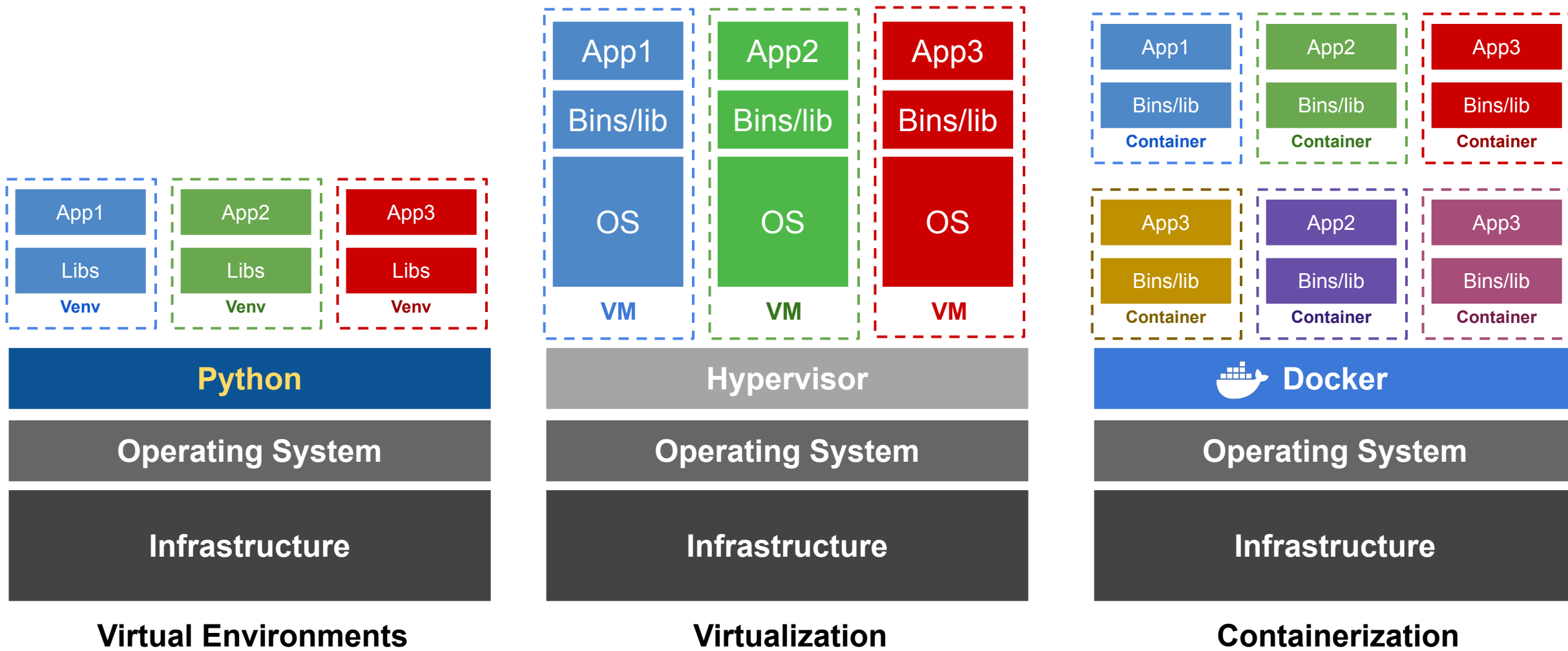
# What is a CONTAINER

---

- Extremely **portable** and lightweight
- **Fully packaged** software with all dependencies included
- Can be used for **development, training, and deployment**
- Development teams can easily **share** containers

**Docker** is an open source platform for building, deploying, and managing containerized applications.

# Environments vs Virtualization vs Containerization



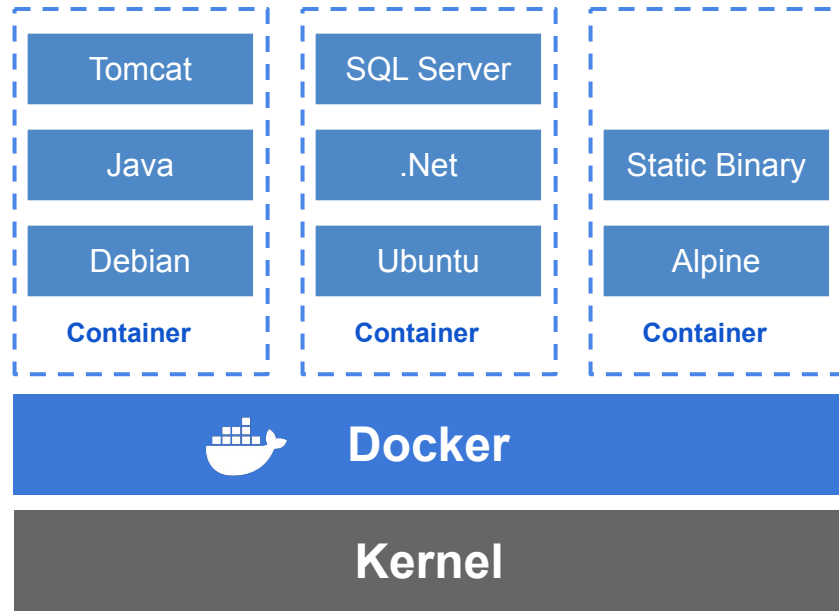
# Tutorial

---

- Let us run the simple-translate app using [Docker](#)
- For this we will do the following:
  - Create a VM Instance
  - SSH into the VM
  - Install Docker inside the VM
  - Run the Containerized simple-translate app
- Full instructions can be found [here](#)



# What is a Container

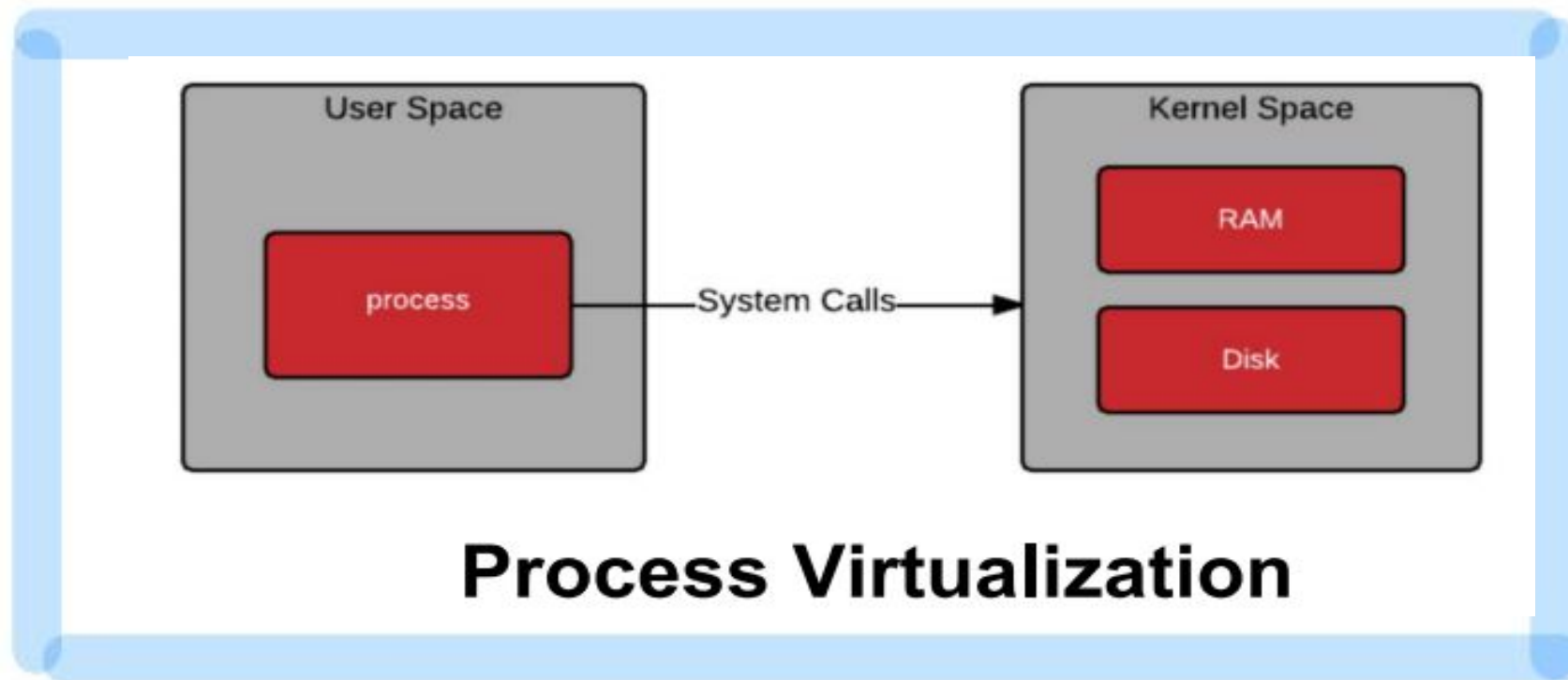


- **Standardized** packaging for software dependencies
- **Isolate** apps from each other
- **Works** for all major Linux distributions, MacOS, Windows

# What Makes Containers so Small?

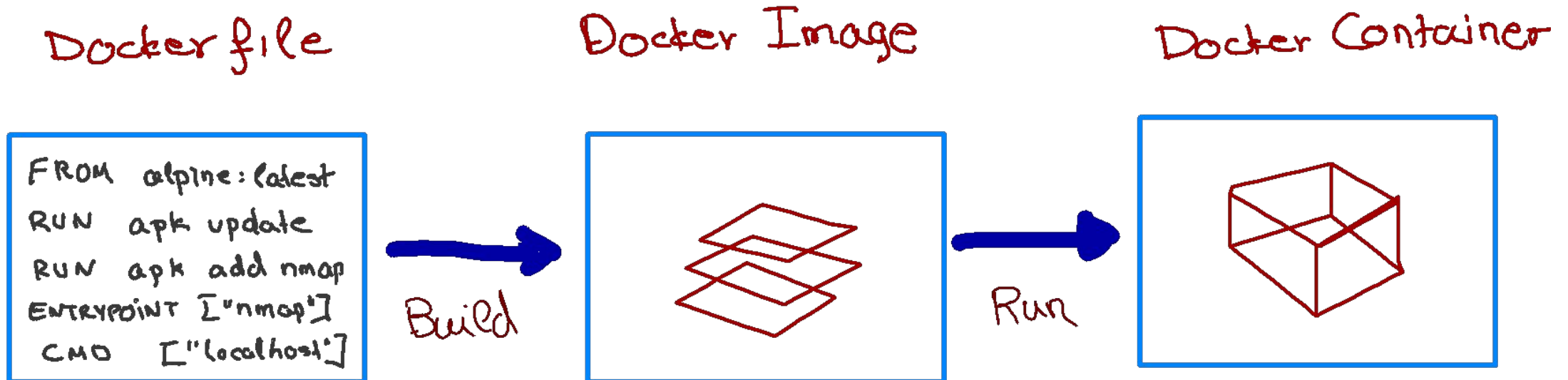
## Container = User Space of OS

- User space refers to all of the code in an operating system that lives outside of the kernel



# How to run a docker container

- We use a simple text file, the Dockerfile, to **build** the Docker Image, which consists of an iso file and other files.
- We **run** the Docker Image to get Docker Container.



# What is the difference between an image and container

---

Docker Image is a template aka a blueprint to create a running Docker container. Docker uses the information available in the Image to create (run) a container.

Image is like a recipe, container is like a dish.

Alternatively, you can think of an image as a class and a container is an instance of that class.

# Inside the Dockerfile

## Dockerfile

```
FROM alpine:latest
RUN apk update
RUN apk add nmap
ENTRYPOINT ["nmap"]
CMD ["localhost"]
```

**FROM:** This instruction in the Dockerfile tells the daemon, which base image to use while creating our new Docker image. In the example here, we are using a very minimal OS image called alpine (just 5 MB of size). You can also replace it with Ubuntu, Fedora, Debian or any other OS image.

**RUN:** This command instructs the Docker daemon to run the given commands as it is while creating the image. A Dockerfile can have multiple RUN commands, each of these RUN commands create a new **layer** in the image.

**ENTRYPOINT:** The ENTRYPOINT instruction is used when you would like your container to run the same executable every time. Usually, ENTRYPOINT is used in scenarios where you want the container to behave exclusively as if it were the executable it's wrapping.

**CMD:** The CMD sets default commands and/or parameters when a docker container runs. **CMD can be overwritten** from the command line via the docker run command.

# Multiple containers from same image

How can you run multiple containers from the same image?

Yes, you could think of an image as instating a class.

Wouldn't they all be identical?

Not necessarily. You could instate it with different parameters using the CMD and therefore different containers will be different.

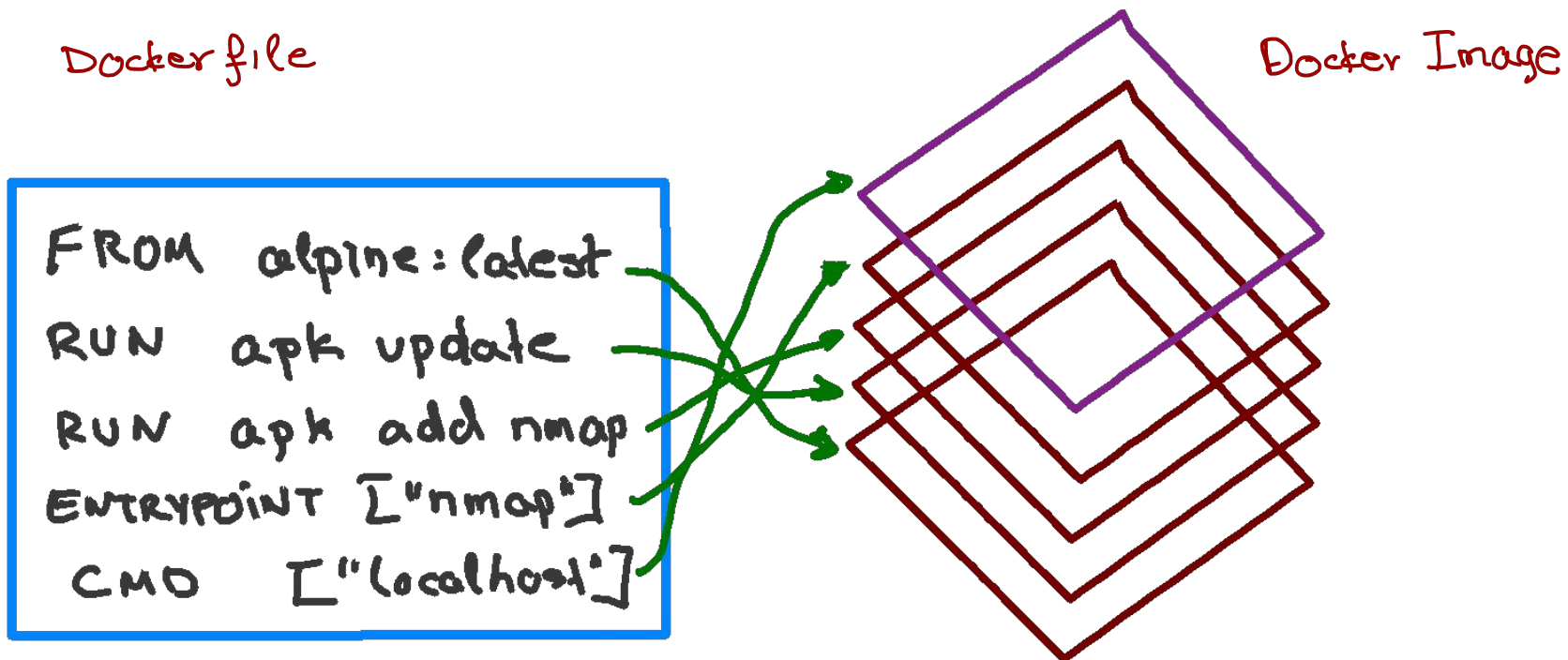
*Dockerfile*

```
FROM ubuntu:latest
RUN apt-get update
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["world"]
```

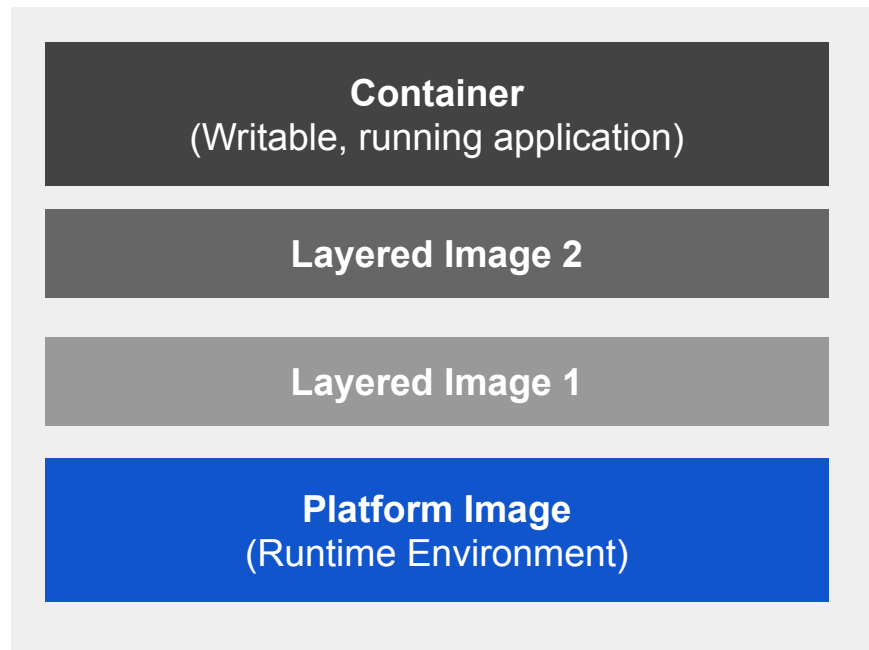
```
> docker build -t hello_world_cmd:first -f Dockerfile_cmd .
> docker run -it hello_world_cmd:first
> Hello world
> docker run -it hello_world_cmd:first Pavlos
> Hello Pavlos
```

# Docker Image as Layers

When we execute the build command, the daemon reads the Dockerfile and creates a layer for every command.



# Image Layering



A application sandbox

- Each container is based on an image that holds necessary config data
- When you launch a container, a writable layer is added on top of the image

A static snapshot of the container configuration

- Layer images are read-only
- Each image depends on one or more parent images

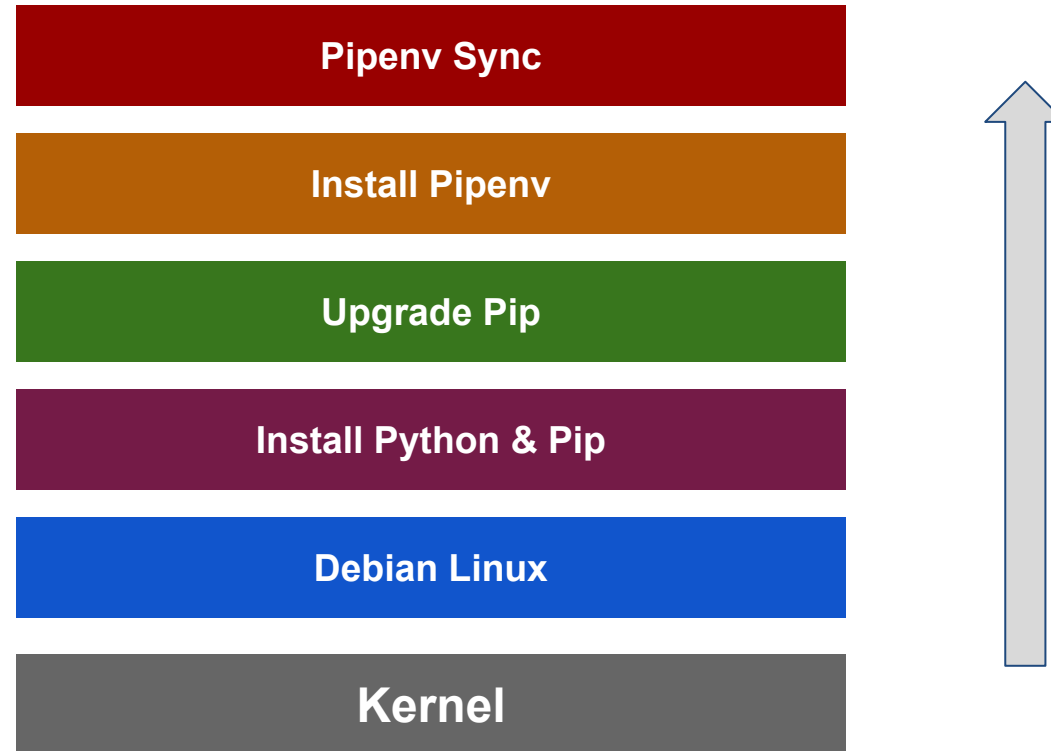
An Image that has no parent

- Platform images define the runtime environment, packages and utilities necessary for containerized application to run

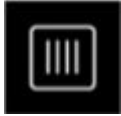


# Image Layering - Example

Docker layers for a container running debian and a python environment using Pipenv



# Some Docker Vocabulary



## **Docker Image**

The basis of a Docker container. Represent a full application



## **Docker Container**

The standard unit in which the application service resides and executes



## **Docker Engine**

Creates, ships and runs Docker containers deployable on a physical or virtual, host locally, in a datacenter or cloud service provider



## **Registry Service (Docker Hub or Docker Trusted Registry)**

Cloud or server-based storage and distribution service for your images

## **Images**

How you **store** your application

## **Containers**

How you **run** your application

# Tutorial

---

## Installing Docker Desktop

1. Install **Docker Desktop**. Use one of the links below to download the proper Docker application depending on your operating system.
  - For Mac users, follow this link- <https://docs.docker.com/docker-for-mac/install/>.
  - For Windows users, follow this link- <https://docs.docker.com/docker-for-windows/install/>  
Note: You will need to install Hyper-V to get Docker to work.
  - For Linux users, follow this link- <https://docs.docker.com/install/linux/docker-ce/ubuntu/>
2. Once installed run the docker desktop.
3. Open a Terminal window and type **docker run hello-world** to make sure Docker is installed properly.

# Tutorial

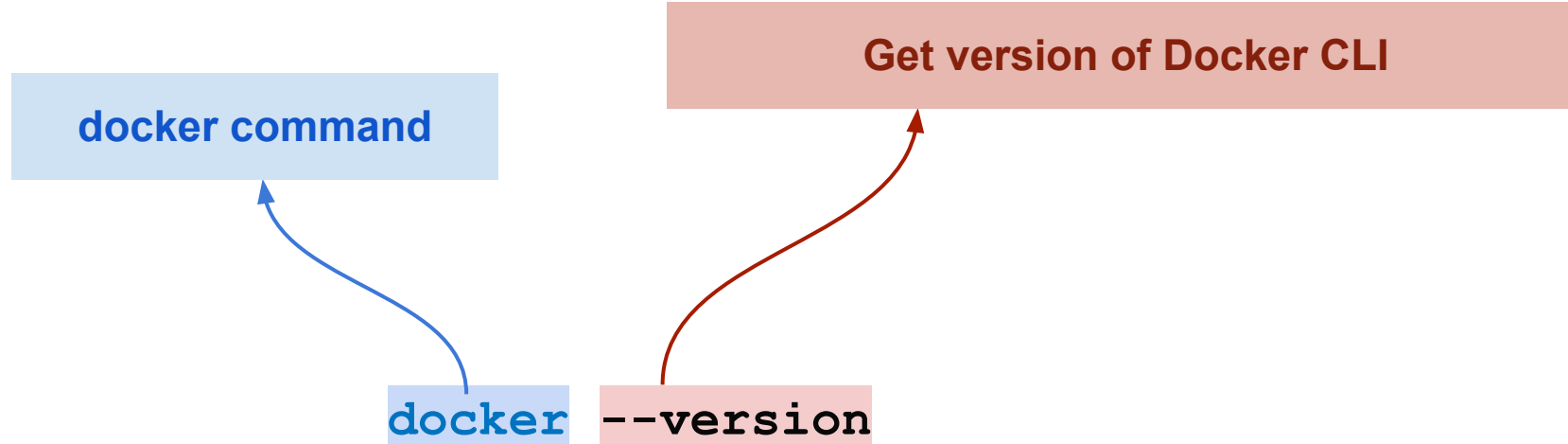
---

- Let us build the simple-translate app [Docker Container](#)
- For this we will do the following:
  - Clone or download [code](#)
  - Build a container
  - Run a container
  - Pavlos will update a container on Docker Hub
  - You will pull the new container and run it
- For detail instruction go [here](#)

# Tutorial: Docker commands

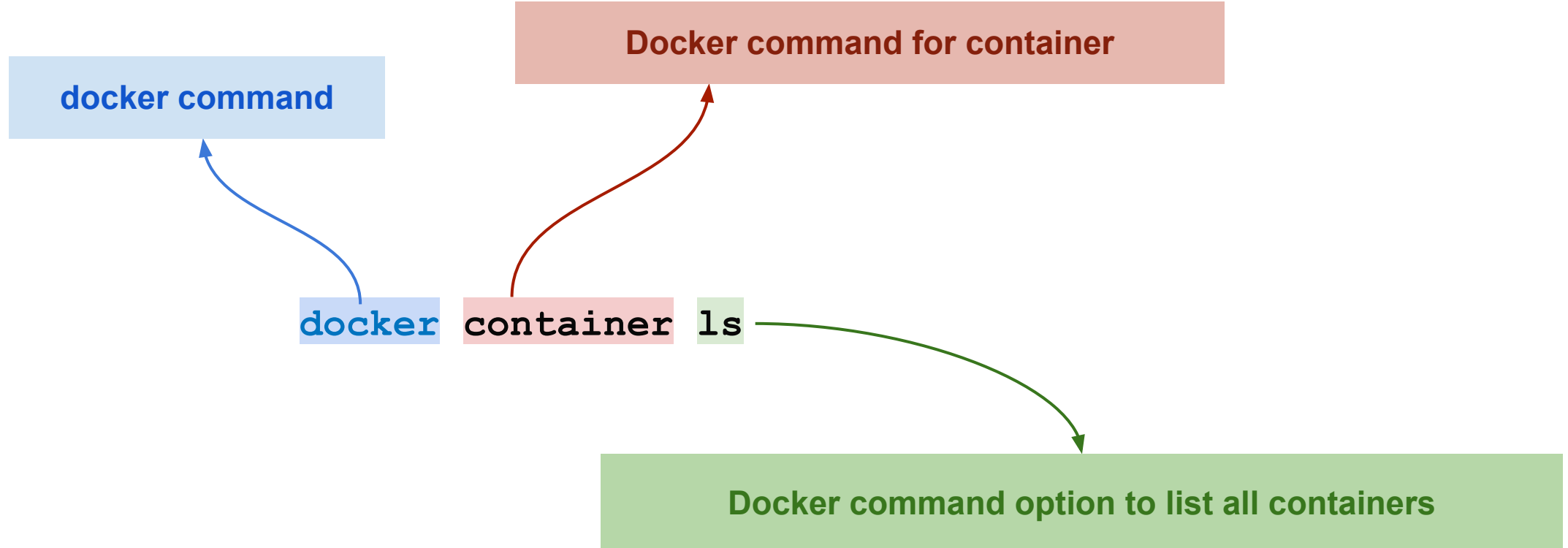
---

Check what version of Docker



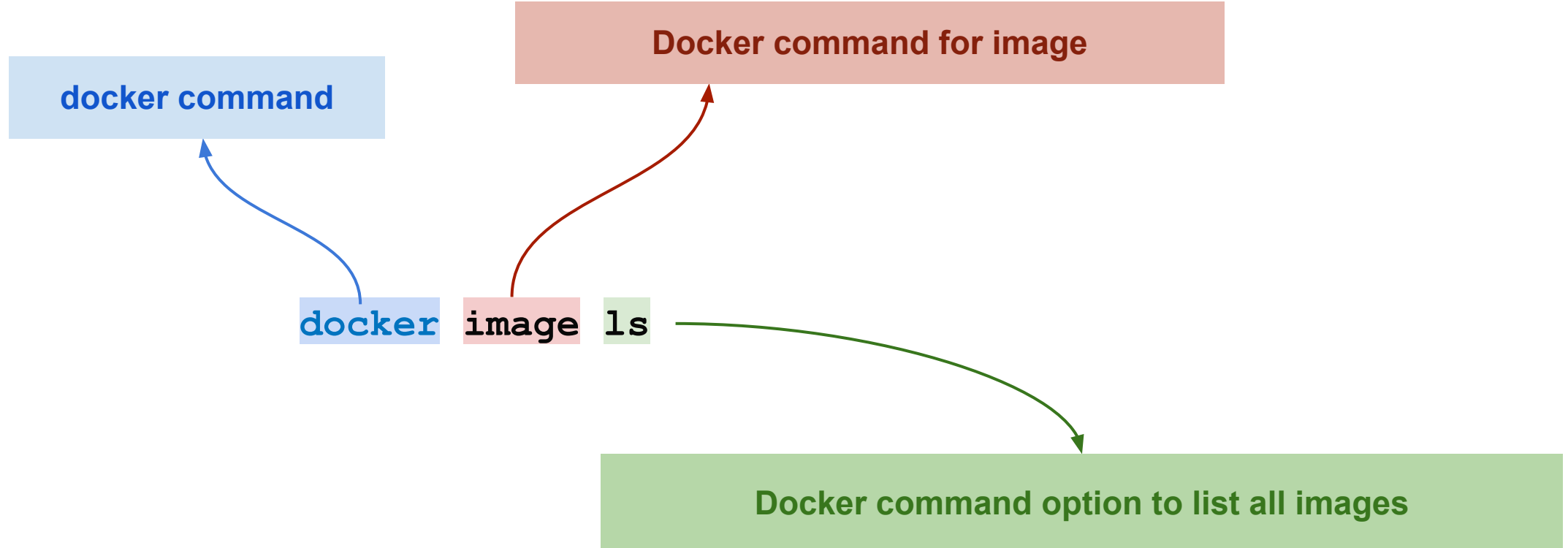
# Tutorial: Docker commands

List all running docker containers



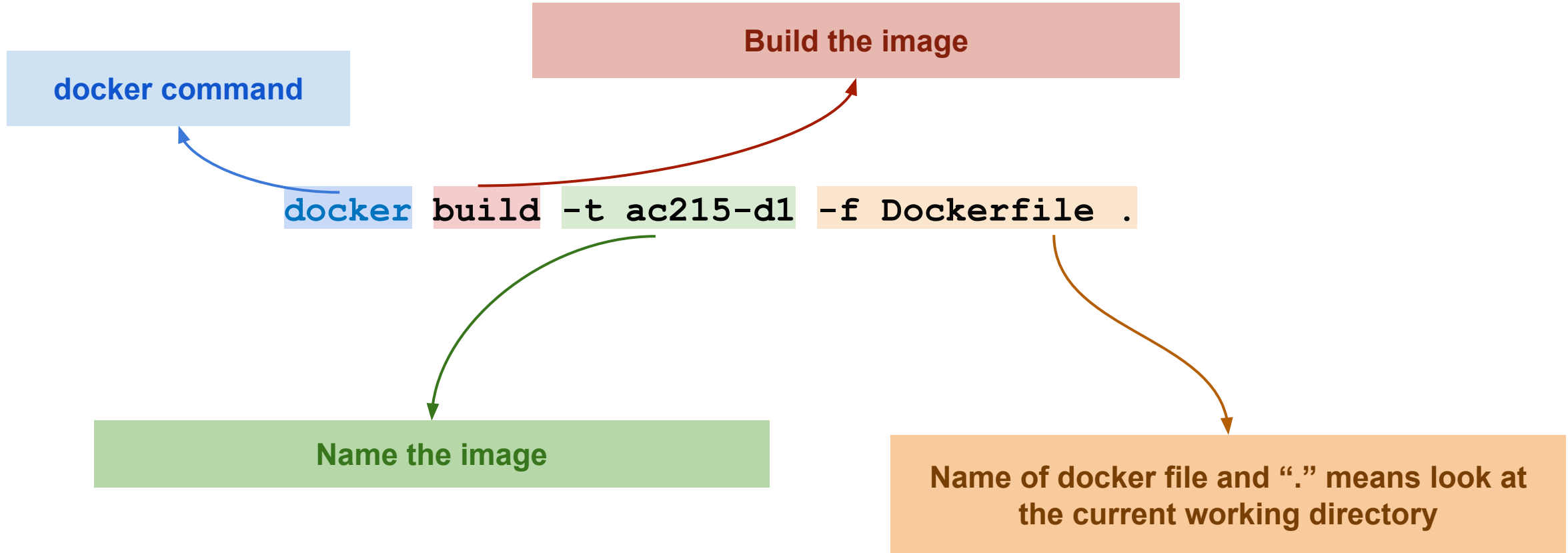
# Tutorial: Docker commands

## List all docker images



# Tutorial: Docker commands

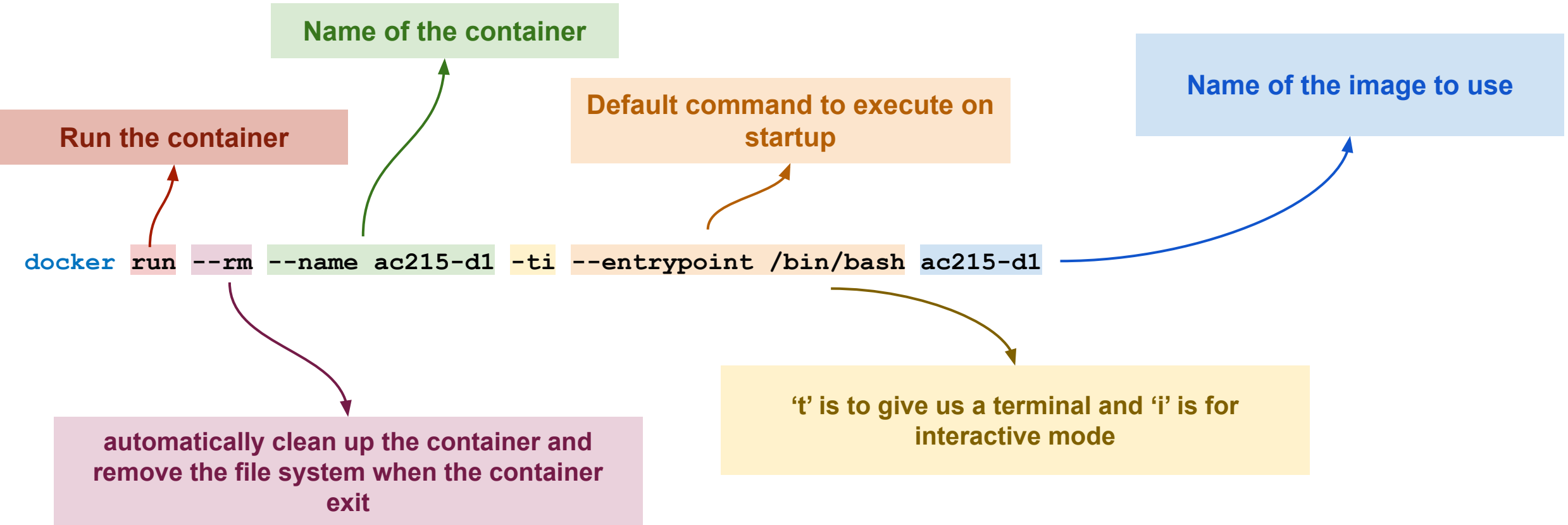
## Build an image based on a Dockerfile





# Tutorial

## Run a docker container using an image from Docker Hub



# Tutorial

---

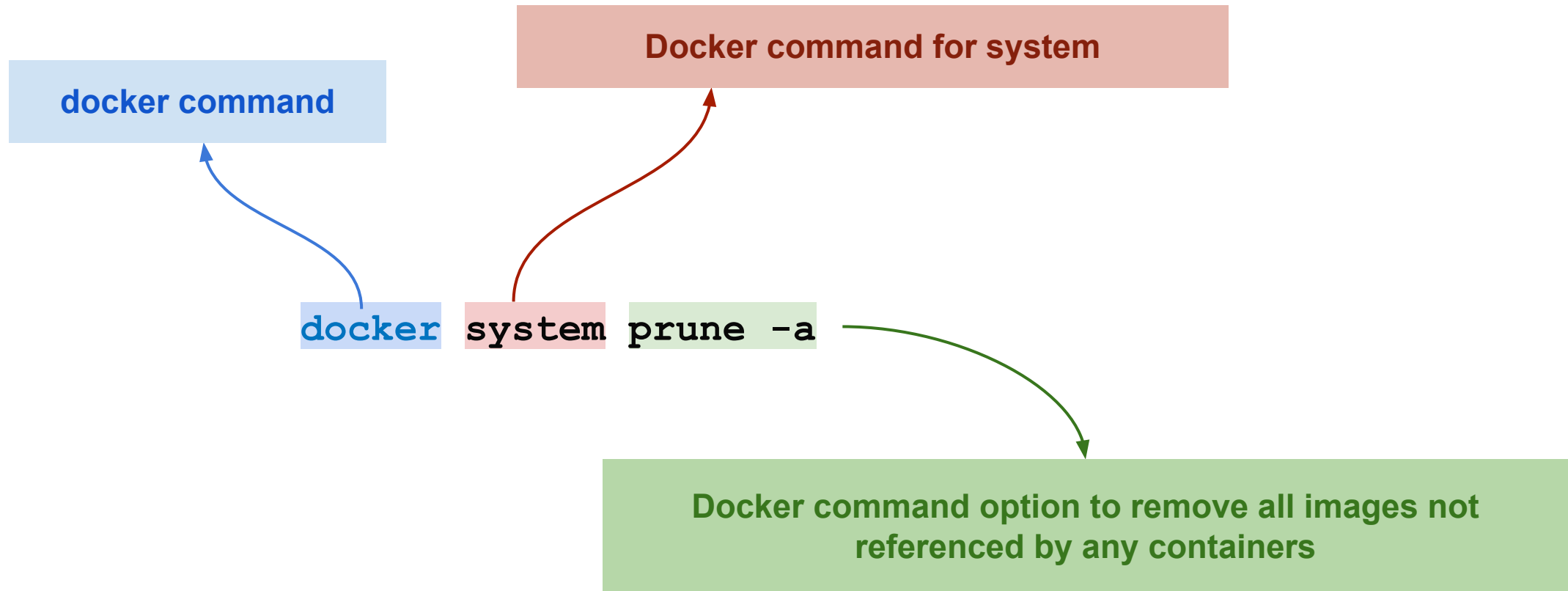
Open another command prompt and check how many container and images we have

```
docker container ls
```

```
docker image ls
```

# Tutorial

Exit from all containers and let us clear of all images



# Tutorial

---

Check how many containers and images we have currently

```
docker container ls
```

```
docker image ls
```

# Docker Image as Layers

```
>docker build -t hello_world_cmd -f Dockerfile_cmd .
```

```
Sending build context to Docker daemon 34.3kB
```

```
Step 1/4 : FROM ubuntu:latest
```

```
latest: Pulling from library/ubuntu
```

```
54ee1f796a1e: Already exists
```

```
f7bfea53ad12: Already exists
```

```
46d371e02073: Already exists
```

```
b66c17bbf772: Already exists
```

```
Digest: sha256:31dfb10d52ce76c5ca0aa19d10b3e6424b830729e32a89a7c6eee2cda2be67a5
```

```
Status: Downloaded newer image for ubuntu:latest
```

```
---> 4e2eef94cd6b
```

```
Step 2/4 : RUN apt-get update
```

```
---> Running in e3e1a87e8d6e
```

```
Get:1 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
```

```
Get:2 http://security.ubuntu.com/ubuntu focal-security InRelease [107 kB]
```

```
Get:3 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [67.5 kB]
```

```
Get:4 http://archive.ubuntu.com/ubuntu focal-updates InRelease [111 kB]
```

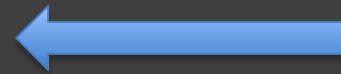
```
Get:5 http://archive.ubuntu.com/ubuntu focal-backports InRelease [98.3 kB]
```

```
Get:6 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [231 kB]
```

```
Get:7 http://archive.ubuntu.com/ubuntu focal/restricted amd64 Packages [33.4 kB]
```

```
Get:8 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages [1275 kB]
```

```
Get:9 http://security.ubuntu.com/ubuntu focal-security/multiverse amd64 Packages [1078 B]
```



Step1: Instruction  
1



Step2: Instruction  
2

# Docker Image as Layers

```
>docker build -t hello_world_cmd -f Dockerfile_cmd .
```

```
....
```

```
Step 3/4 : ENTRYPOINT ["/bin/echo", "Hello"]
```

```
---> Running in 52c7a98397ad
```

```
Removing intermediate container 52c7a98397ad
```

```
---> 7e4f8b0774de
```

```
Step 4/4 : CMD ["world"]
```

```
---> Running in 353adb968c2b
```

```
Removing intermediate container 353adb968c2b
```

```
---> a89172ee2876
```

```
Successfully built a89172ee2876
```

```
Successfully tagged hello_world_cmd:latest
```



Step3: Instruction 3



Step4: Instruction 4

# Docker Image as Layers

```
> docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello_world_cmd	latest	a89172ee2876	7 minutes ago	96.7MB
ubuntu	latest	4e2eef94cd6b	3 weeks ago	73.9MB

```
> docker image history hello_world_cmd
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
a89172ee2876	8 minutes ago	/bin/sh -c #(nop) CMD ["world"]	0B	
7e4f8b0774de	8 minutes ago	/bin/sh -c #(nop) ENTRYPOINT ["/bin/echo" "...	0B	
cfc0c414a914	8 minutes ago	/bin/sh -c apt-get update	22.8MB	
4e2eef94cd6b	3 weeks ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	0B	
<missing>	3 weeks ago	/bin/sh -c mkdir -p /run/systemd && echo 'do...	7B	
<missing>	3 weeks ago	/bin/sh -c set -xe && echo '#!/bin/sh' > /...	811B	
<missing>	3 weeks ago	/bin/sh -c [ -z "\$(apt-get indextargets)" ]	1.01MB	
<missing>	3 weeks ago	/bin/sh -c #(nop) ADD file:9f937f4889e7bf646...	72.9MB	

# Why Layers

Why build an image with multiple layers when we can just build it in a single layer?

Let's take an example to explain this concept better, let us try to change the Dockerfile\_cmd we created and rebuild a new Docker image.

```
> docker build -t hello_world_cmd -f Dockerfile_cmd .
```

```
Sending build context to Docker daemon 34.3kB
```

```
Step 1/4 : FROM ubuntu:latest
```

```
---> 4e2eef94cd6b
```

```
Step 2/4 : RUN apt-get update
```

```
---> Using cache
```

```
---> cfc0c414a914
```

```
Step 3/4 : ENTRYPOINT ["/bin/echo", "Hello"]
```

```
---> Using cache
```

```
---> 7e4f8b0774de
```

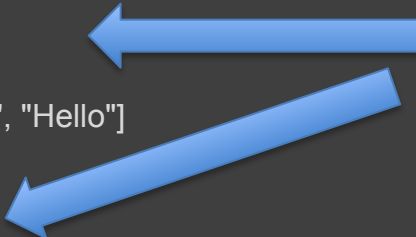
```
Step 4/4 : CMD ["world"]
```

```
---> Using cache
```

```
---> a89172ee2876
```

```
Successfully built a89172ee2876
```

```
Successfully tagged hello_world_cmd:latest
```



Have seen this before. Use  
cache

As you can see that the image was built using the **existing** layers from our previous docker image builds. If some of these layers are being used in **other containers**, they can just use the existing layer instead of recreating it from scratch.

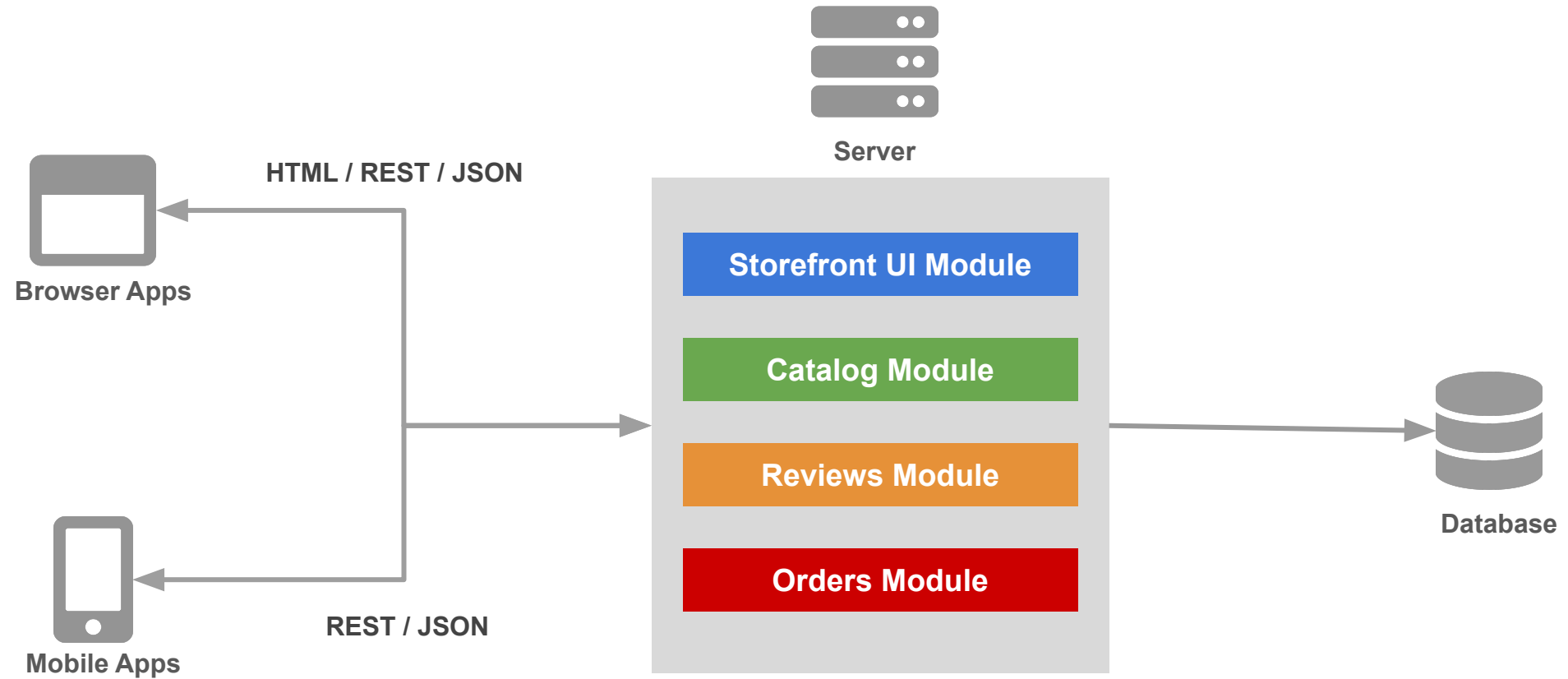


# Why use Containers?

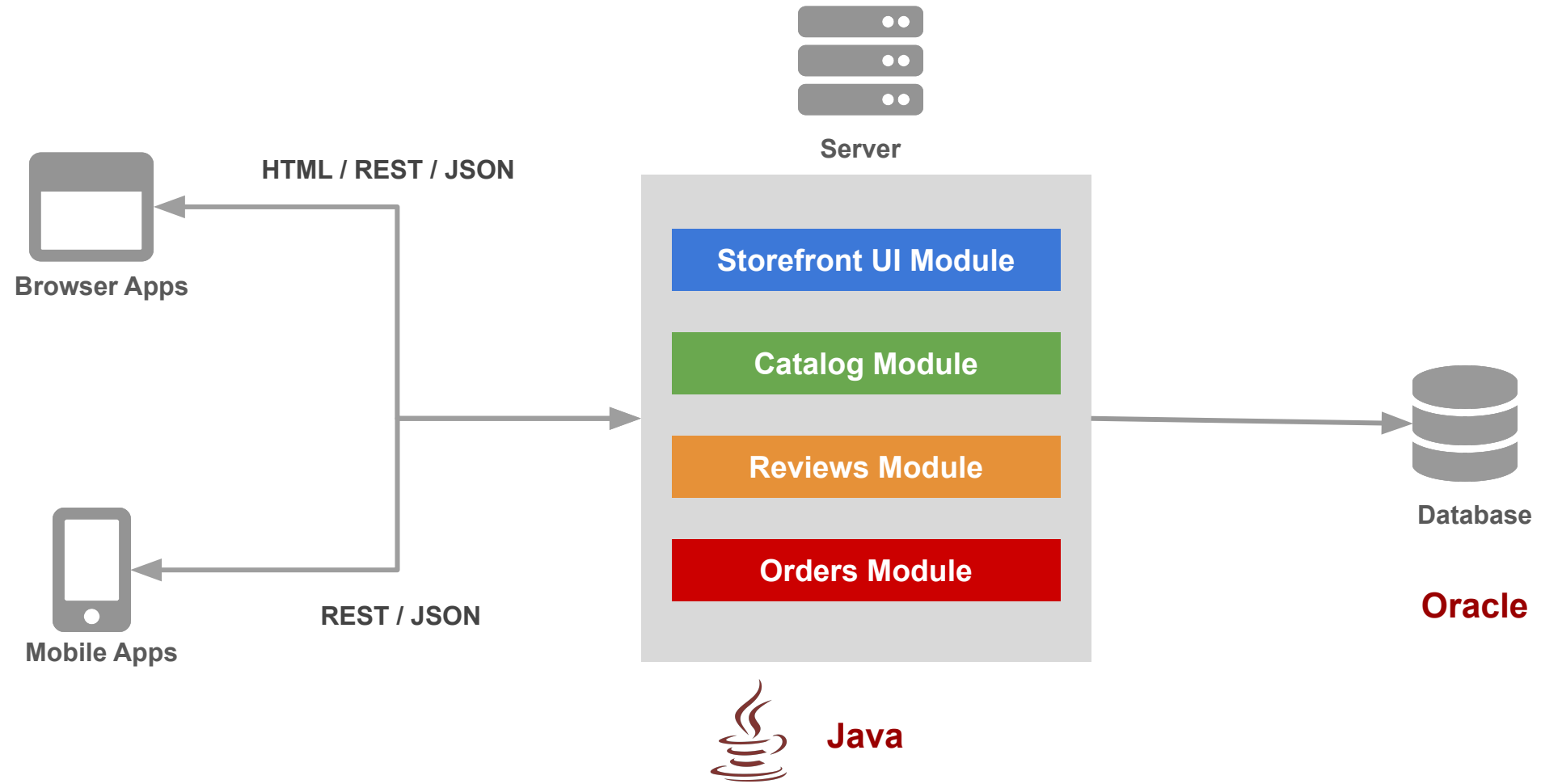
---

- Imagine you are building a large complex application (e.g. Online Store)
- Traditionally you would build this using a **Monolithic Architecture**

# Monolithic Architecture



# Monolithic Architecture



# Monolithic Architecture - Advantages

---

Simple to **Develop, Test, Deploy** and **Scale**:

1. Simple to develop because all the tools and IDEs support the applications by default
2. Easy to deploy because all components are packed into one bundle
3. Easy to scale the whole application

# Monolithic Architecture - Disadvantages

---

1. Very difficult to maintain
2. One component failure will cause the whole system to fail
3. Very difficult to create the patches for monolithic architecture
4. Adapting to new technologies is challenging
5. Take a long time to startup because all the components needs to get started

# Applications have changed dramatically

---

## **A decade ago**

Apps were monolithic  
Built on a single stack (e.e. .NET or Java)  
Long lived  
Deployed to a single server

## **Today**

Apps are constantly being developed  
Build from loosely coupled components  
Newer version are deployed often  
Deployed to a multitude of servers

# Applications have changed dramatically

---

## **A decade ago**

Apps were monolithic  
Built on a single stack (e.e. .NET or Java)  
Long lived  
Deployed to a single server

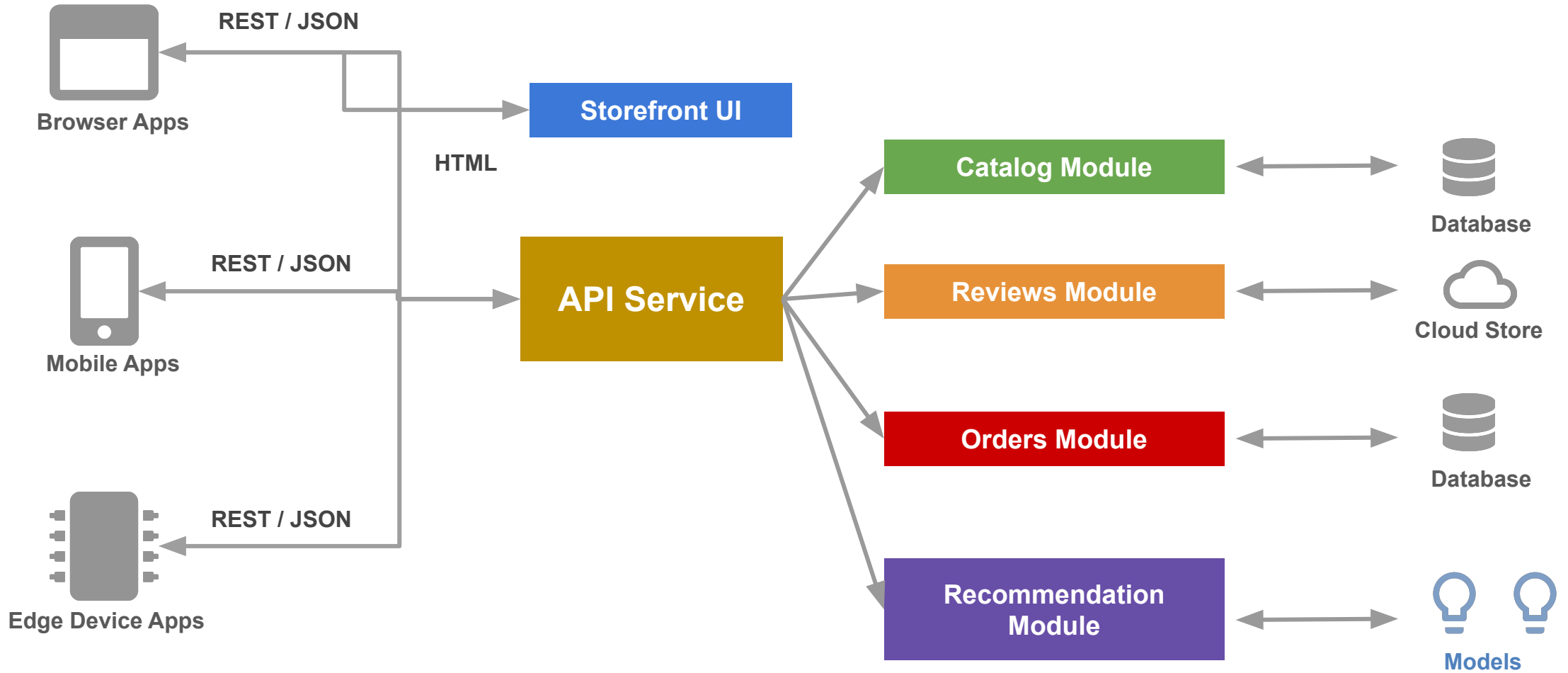
## **Today**

Apps are constantly being developed  
Build from loosely coupled components  
Newer version are deployed often  
Deployed to a multitude of servers

## **Data Science**

**Apps are being integrated with various data types/sources and models**

# Today: Microservice Architecture





# Software Development Workflow (no Docker)

## Windows



Node.js  
Python



## Linux



Node.js  
Python



## Mac

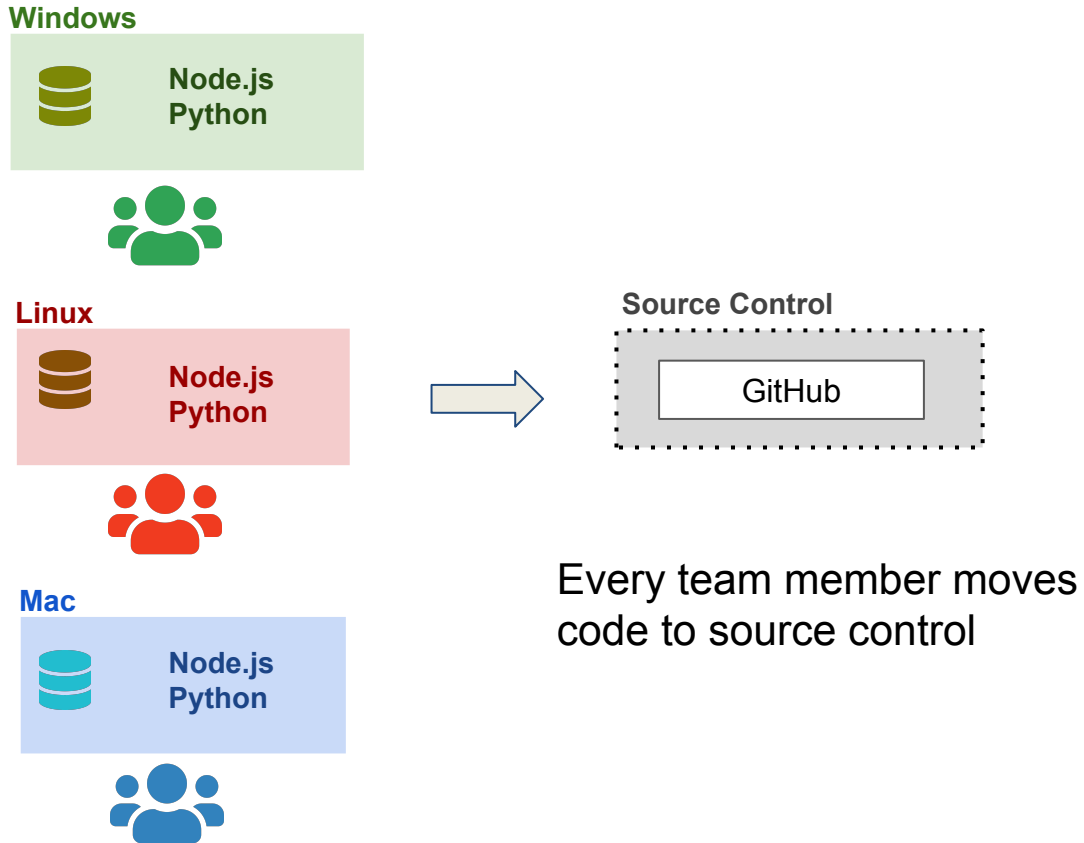


Node.js  
Python



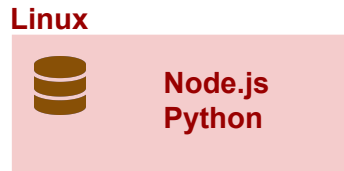
OS Specific **installation** in  
every developer machine

# Software Development Workflow (no Docker)

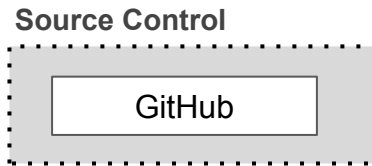


OS Specific **installation** in every developer machine

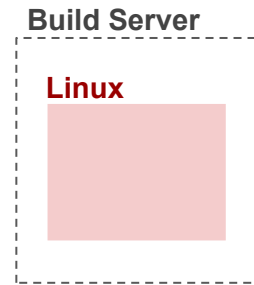
# Software Development Workflow (no Docker)



OS Specific **installation** in every developer machine



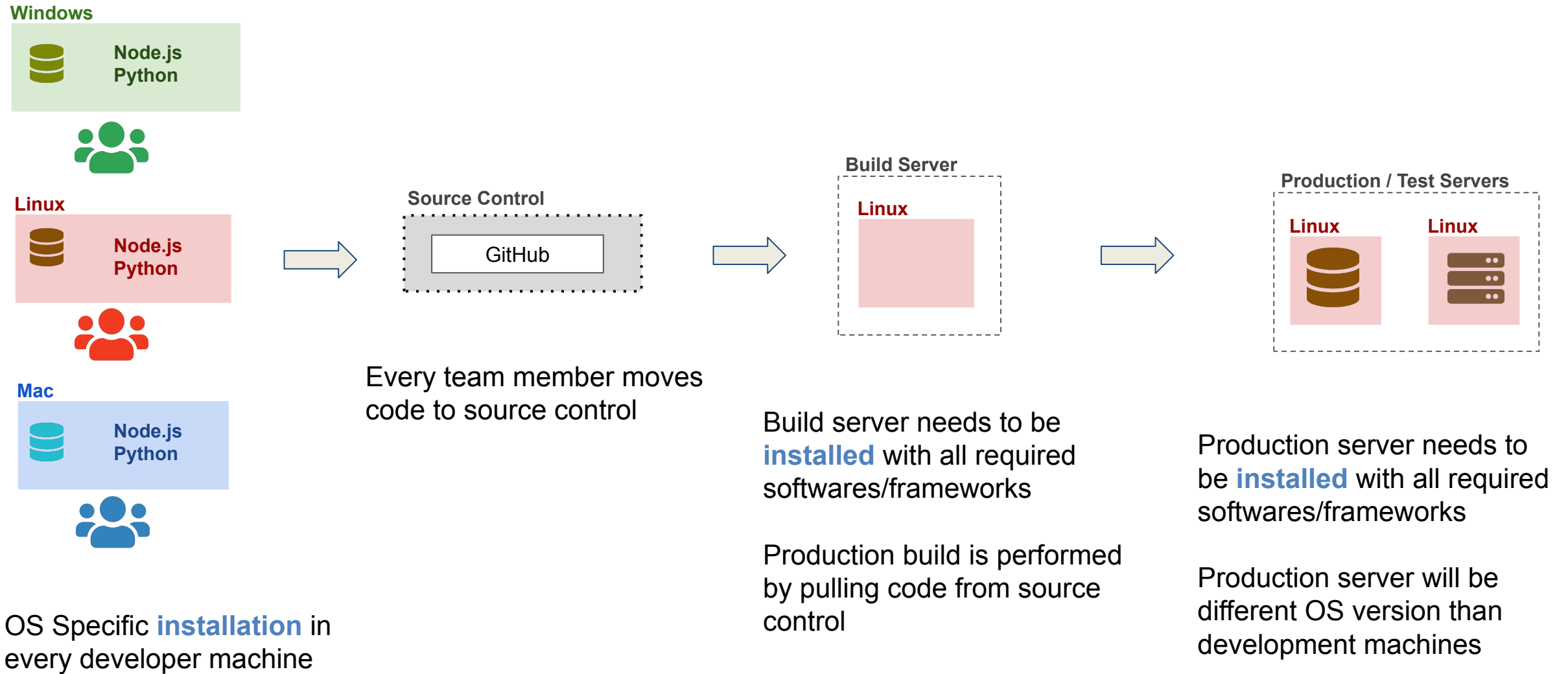
Every team member moves code to source control



Build server needs to be **installed** with all required softwares/frameworks

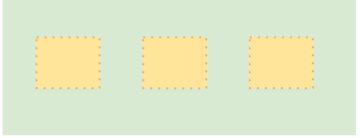
Production build is performed by pulling code from source control

# Software Development Workflow (no Docker)



# Software Development Workflow (with Docker)

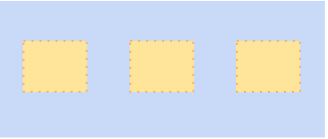
Windows



Linux



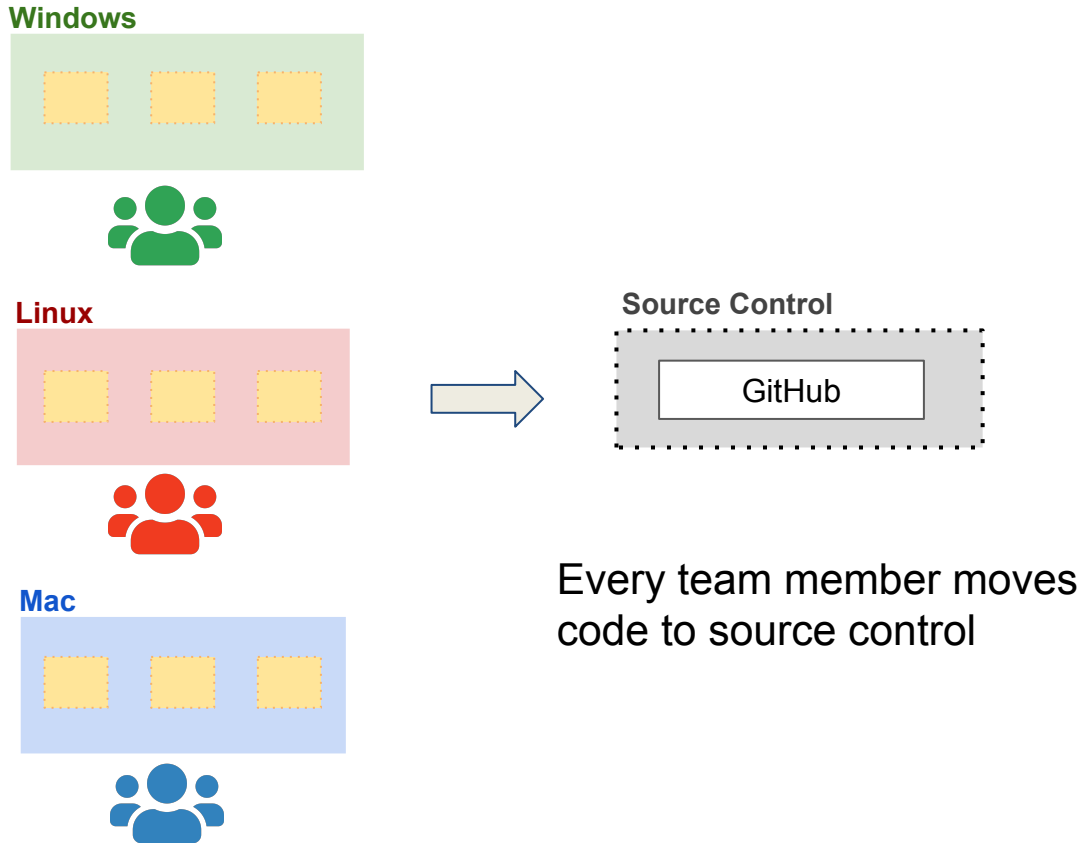
Mac



Development machines only  
needs **Docker installed**

**Containers** need to be setup  
only once

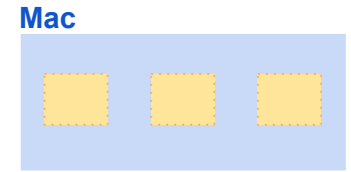
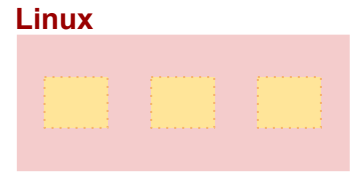
# Software Development Workflow (with Docker)



Development machines only  
needs **Docker installed**

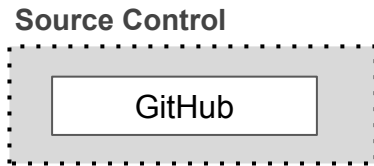
**Containers** need to be setup  
only once

# Software Development Workflow (with Docker)

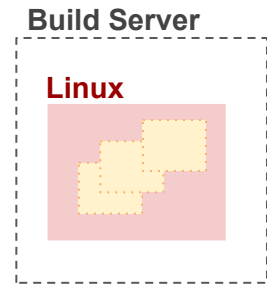


Development machines only needs **Docker installed**

**Containers** need to be setup only once



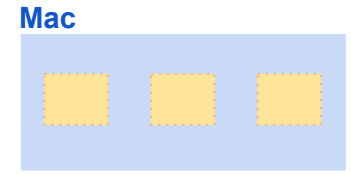
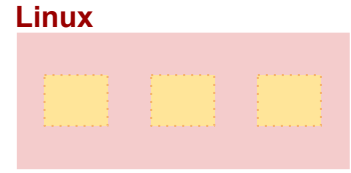
Every team member moves code to source control



Build server only needs **Docker installed**

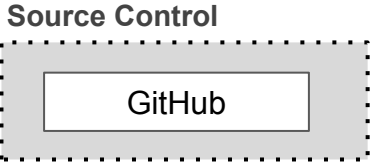
Docker **images** are built for a release and pushed to **container registry**

# Software Development Workflow (with Docker)

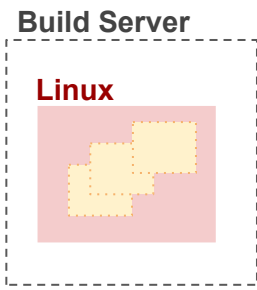
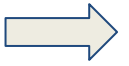


Development machines only needs **Docker installed**

**Containers** need to be setup only once

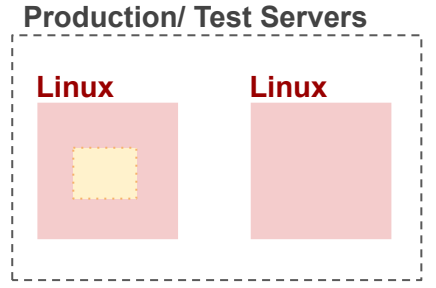


Every team member moves code to source control



Build server only needs **Docker installed**

Docker **images** are built for a release and pushed to **container registry**



Production server only needs **Docker installed**

Production server pulls Docker **images** from **container registry** and runs them



# Comparison

	VIRTUAL ENV	DOCKER	VM	JH
COMPUTATIONAL COST MEMORY FOOTPRINT	LOW	MEDIUM LOW	HIGH	?
DEPLOYMENT	EASY	MEDIUM	SWIT HIGH THEN EASY	N/A
VERSATILITY (TYPES OF APPS)	MEDIUM	MEDIUM HIGH	MEDIUM HIGH	LOW
PORTABILITY	MEDIUM	HIGH	HIGH	HIGH

- COMPUTATIONAL SCIENCE
- DEV OPS
- DATA SCIENCE (NO PIPELINES)
- DATA SCIENCE (PIPELINES)



**THANK YOU**