Intro to Docker

How to Containerize Applications

Delivered by: Noureddin Sadawi, PhD

About the Speaker

Name: Dr Noureddin Sadawi

Qualification: PhD Computer Science

Experience: Several areas including but not limited to Docker, Machine Learning and Data Science, Python and much more.

What is Docker? 1/2

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.

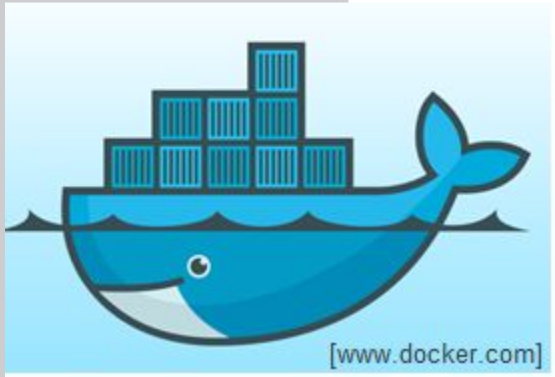
[Source:https://opensource.com/resources/what-docker]

What is Docker? 2/2

- Provides a uniformed wrapper around a software package: *(Build, Ship and Run Any App, Anywhere)*

  [www.docker.com]

- Similar to shipping real containers: The container is always the same, regardless of the contents and thus fits on all trucks, cranes, ships, ...



What are Containers? 1/2

- The industry standard is/was to use Virtual Machines (VMs) to run software applications.
- VMs run applications inside a guest Operating System, which runs on virtual hardware powered by the host OS.
- VMs are great at providing <u>full process isolation for applications</u>: there are very few ways a problem in the host operating system can affect the software running in the guest operating system, and vice-versa.

What are Containers? 2/2

- But this isolation comes at great cost — the computational overhead spent virtualizing hardware for a guest OS to use is substantial.
- Containers take a different approach: by leveraging the low-level mechanics of the host operating system, containers provide most of the isolation of virtual machines at a fraction of the computing power.

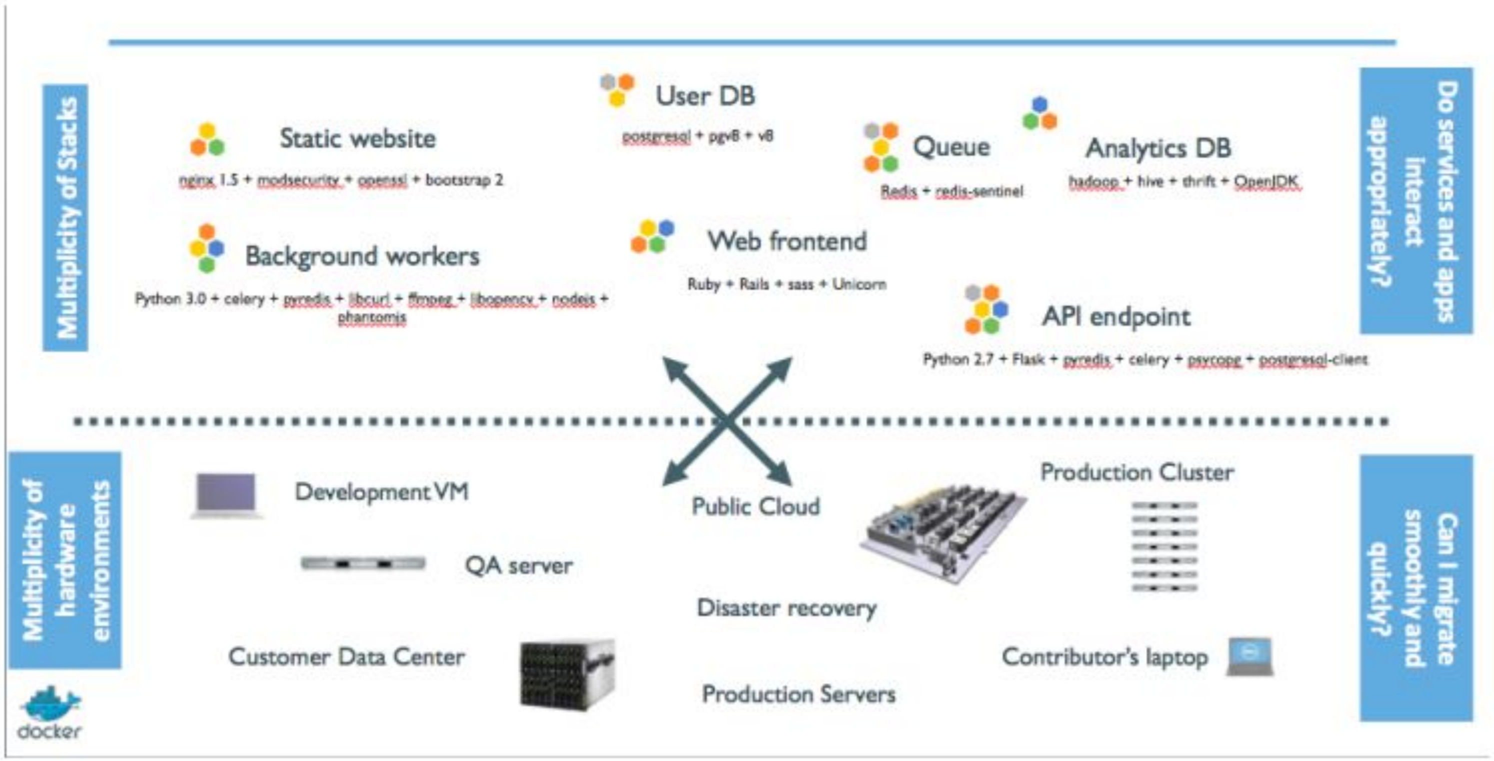
Why use Containers? 1/2

- Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run.
- This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer’s personal laptop.
- This gives developers the ability to create predictable environments that are isolated from the rest of the applications and can be run anywhere (dependency hell).

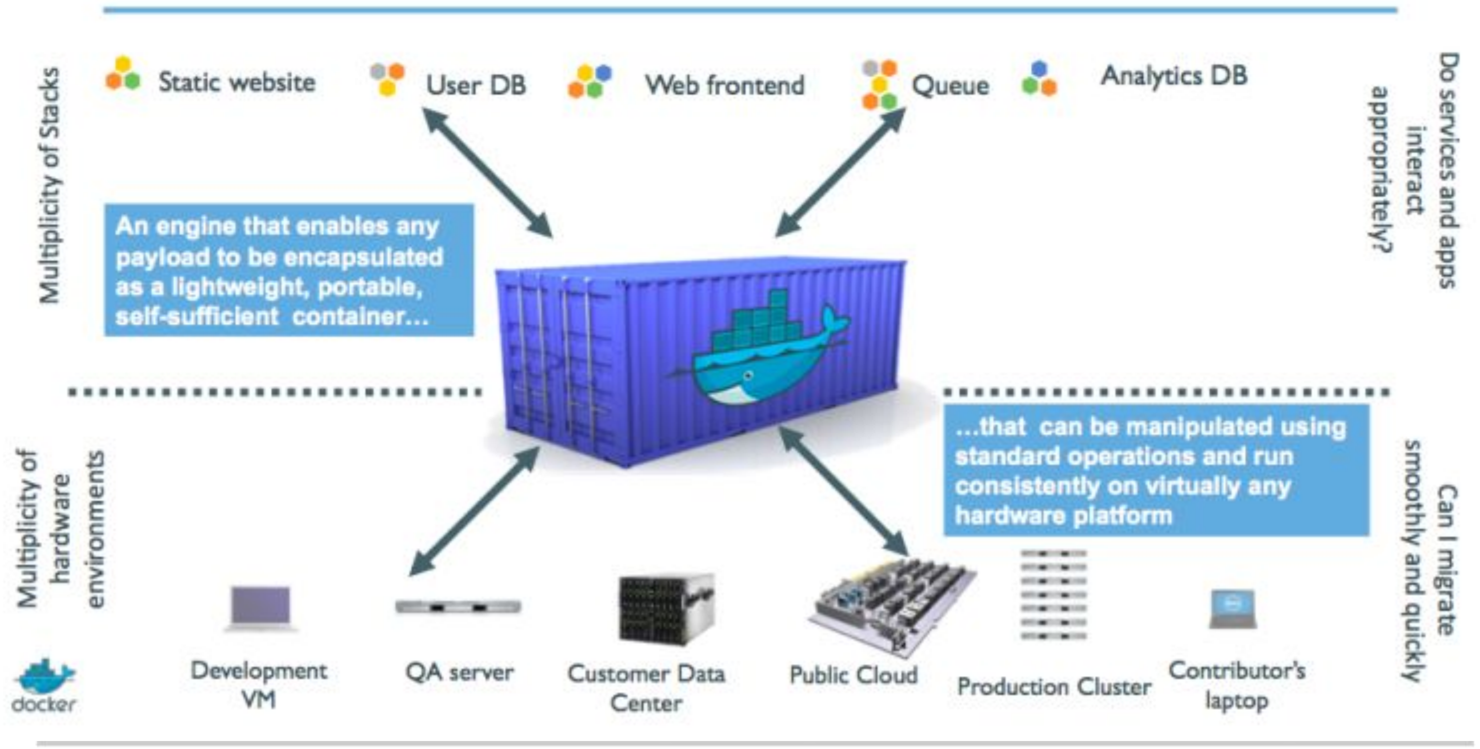
Why use Containers? 2/2

- From an operations standpoint, apart from portability containers also give more granular control over resources giving your infrastructure improved efficiency which can result in better utilization of your compute resources.
- Due to these benefits, containers (& Docker) have seen widespread adoption.
- Companies like Google, Facebook and others leverage containers to make large engineering teams more productive and to improve utilization of compute resources.

The Problem Before Docker

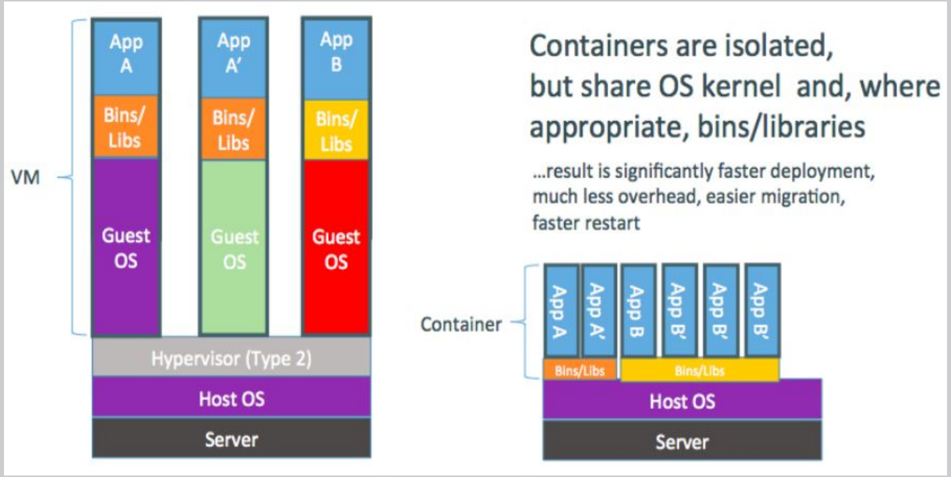

Multiplicity of Stacks
Static website
nginx 1.5 + modsecurity + openssl + bootstrap 2
User DB
postgresql + pgv8 + v8
Queue
Redis + redis-sentinel
Analytics DB
hadoop + hive + thrift + OpenJDK
Background workers
Python 3.0 + celery + pyredis + libcurl + ffmpeg + libopencv + nodejs + phantomjs
Web frontend
Ruby + Rails + sass + Unicorn
API endpoint
Python 2.7 + Flask + pyredis + celery + psycopg + postgresql-client
Do services and apps interact appropriately?
Multiplicity of hardware environments
Development VM
QA server
Public Cloud
Production Cluster
Disaster recovery
Customer Data Center
Contributor's laptop
Production Servers
Can I migrate smoothly and quickly?
docker

After Docker


Multiplicity of Stacks
Static website
User DB
Web frontend
Queue
Analytics DB
Do services and apps interact appropriately?
An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…
…that can be manipulated using standard operations and run consistently on virtually any hardware platform
Multiplicity of hardware environments
Development VM
QA server
Customer Data Center
Public Cloud
Production Cluster
Contributor's laptop
Can I migrate smoothly and quickly
docker

VMs vs Containers 1/2

- **VMs:** Each virtualized application includes not only the application - which may be only 10s of MBs - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.



VM
App A
Bins/ Libs
Guest OS
App A'
Bins/ Libs
Guest OS
App B
Bins/ Libs
Guest OS
Hypervisor (Type 2)
Host OS
Server
Containers are isolated, but share OS kernel and, where appropriate, bins/libraries
...result is significantly faster deployment, much less overhead, easier migration, faster restart
Container
App A
App A'
App B
App B'
App B'
App B'
Bins/Libs
Bins/Libs
Host OS
Server

VMs vs Containers 2/2

- **Docker:** The container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers.
- Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.
- Docker provides base images that contain OS installations we can start from: The OS is not more than an application running on the Kernel...

Commoditization of Containers

Container technology has been around for a while ... <u>So what's new?</u>

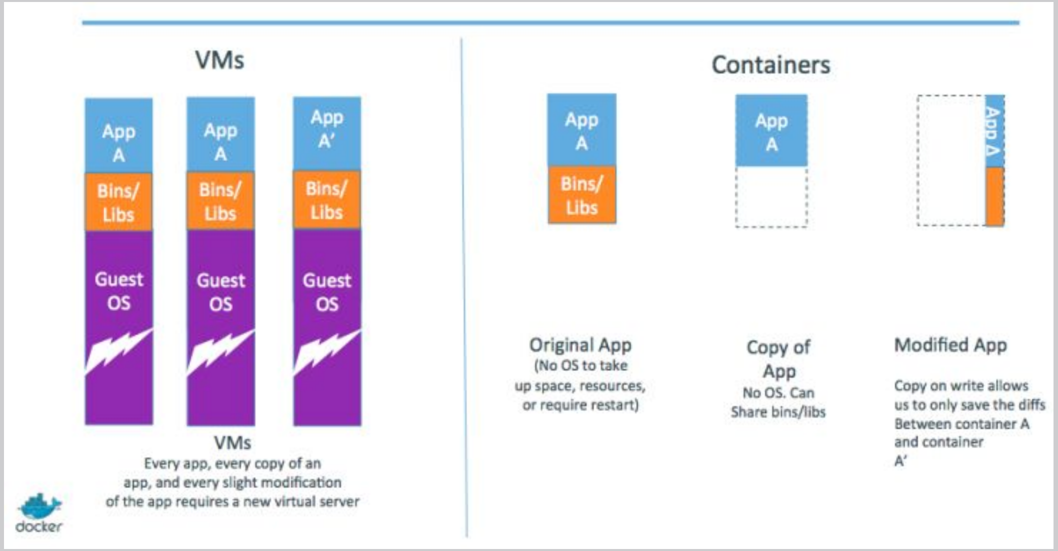
- Standardize the container format, because containers were not portable.
- Analogy:
  - Shipping containers are not just steel boxes .. they are steel boxes of a standard size, with the same hooks and holes.
- Make containers easy to use for developers.
- Emphasis on reusable components, APIs, ecosystem of standard tools.

Running Containers Everywhere

- Maturity of underlying technology.
- Ability to run on any [Linux] server today: physical, virtual, VM, cloud, OpenStack…
- Ability to move between any of the above in a matter of seconds with no modification or delay.
- Ability to share containerized components.
- Self contained environment - no <u>dependency hell.</u>
- Tools for how containers work together: linking, discovery, orchestration...

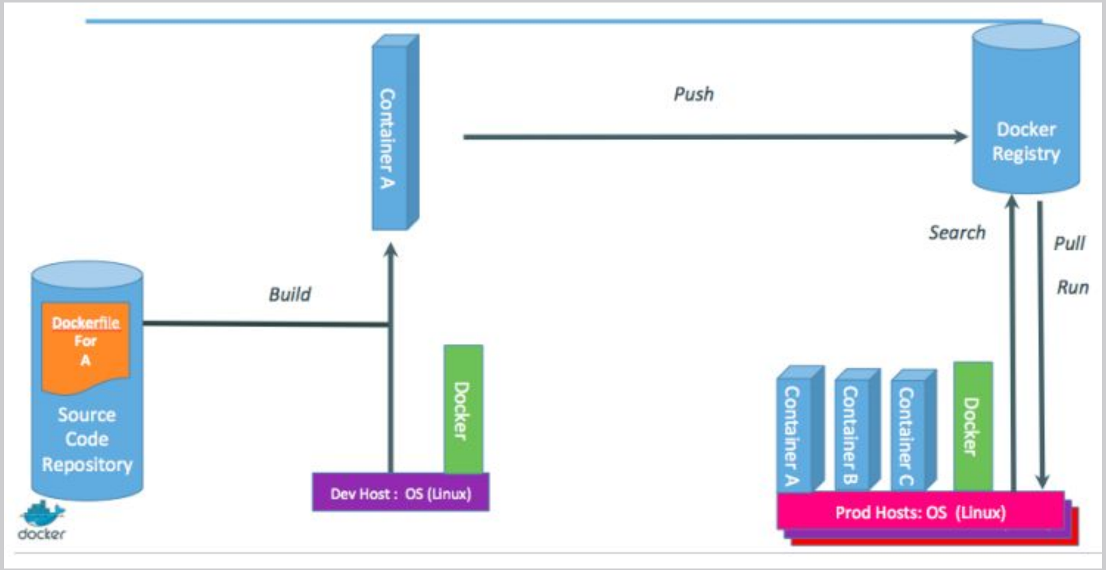
Shipping Containers Efficiently

- Ship container images, made of reusable shared layers.
- Optimizes disk usage, memory usage, network usage.


VMs
App A
Bins/ Libs
Guest OS
App A
Bins/ Libs
Guest OS
App A'
Bins/ Libs
Guest OS
VMs
Every app, every copy of an app, and every slight modification of the app requires a new virtual server
Containers
App A
Bins/ Libs
Original App
(No OS to take up space, resources, or require restart)
App A
Copy of App
No OS. Can Share bins/libs
App A'
Modified App
Copy on write allows us to only save the diffs Between container A and container A'
docker

Containers in a Modern Software Factory

- The container becomes the new build artefact.
- The same container can go from dev, to test, to QA, to prod.

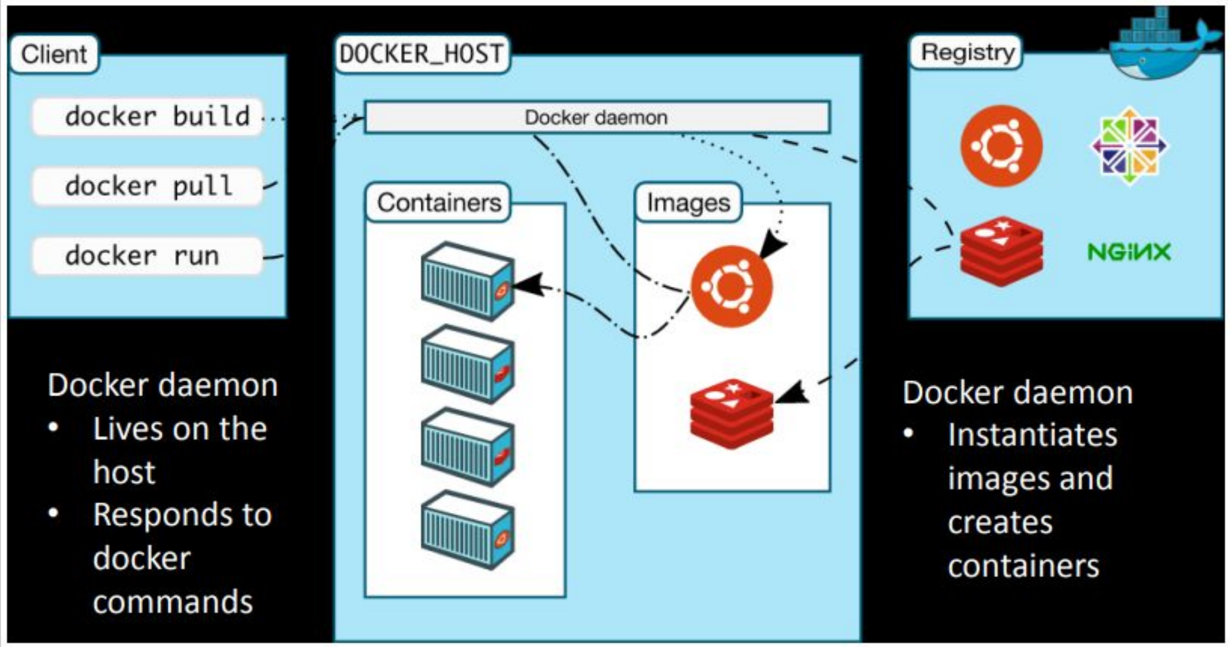


Docker Architecture

Docker is a client-server application:

- The Docker daemon:
  - Receives and processes incoming Docker API requests.
- The Docker client:
  - Command line tool - the docker binary.
  - Talks to the Docker daemon via the Docker API.
- Docker Hub Registry:
  - Public image registry.
  - The Docker daemon talks to it via the registry API.

Docker Architecture



Docker daemon
- Lives on the host
- Responds to docker commands

Docker daemon
- Instantiates images and creates containers

A container is formed by instantiating an image

Typical Situations for using Docker

1. To Simplify Configuration.
2. To Manage Code Pipeline(s).
3. To increase Developer Productivity.
4. For App Isolation.
5. To enhance Debugging Capabilities.
6. For better Multi-tenancy.
7. Rapid Deployment.

Microservices 1/2

Concepts

1. The idea is to divide an application into smaller applications (components or services).
2. Services should be minimal and complete.
3. Services should be interchangeable.
4. Services should communicate through language agnostic APIs.

Microservices 2/2

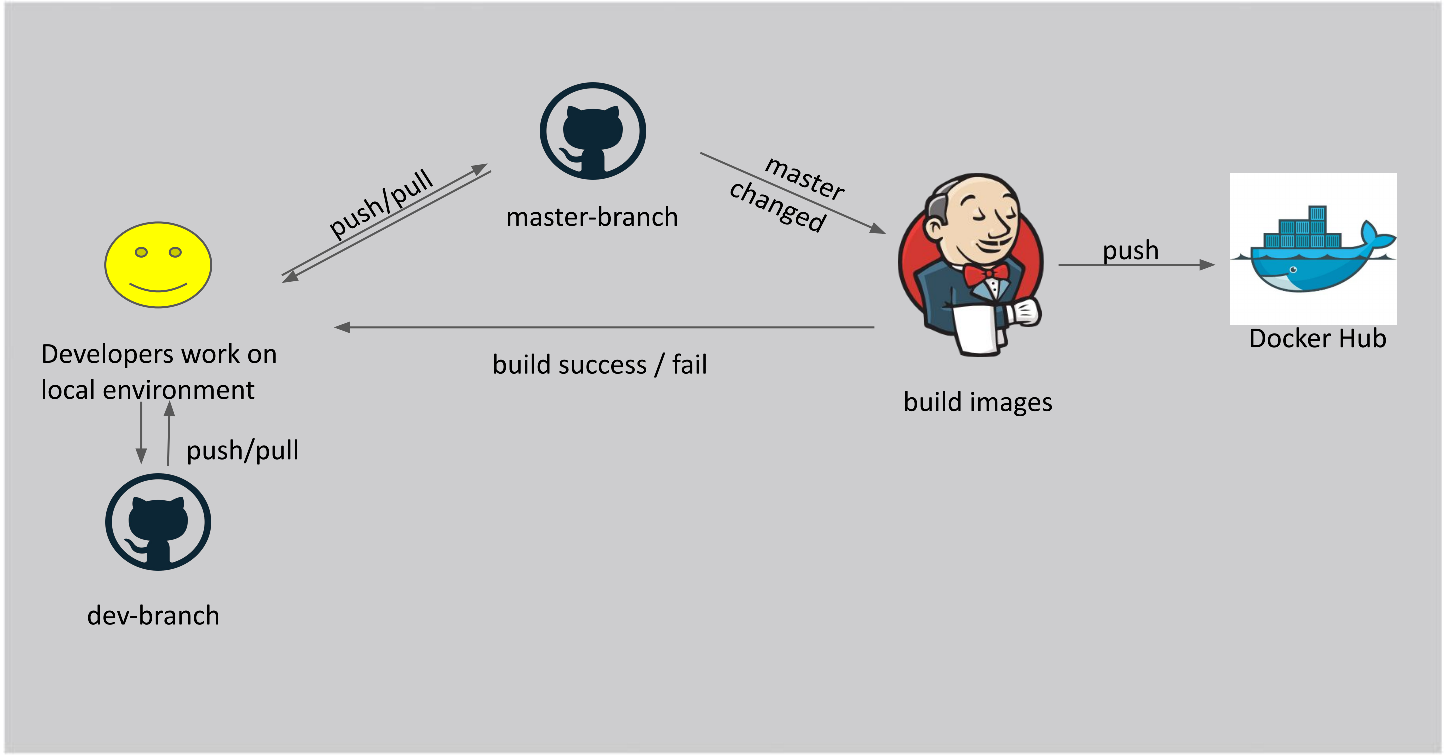
Pros

- Separation of concerns (can deal with components separately).
- Suitable for Unit Testing.
- Reusability (the same component can be used in multiple applications).
- Resilience (if a component fails, a replacement should be run asap).

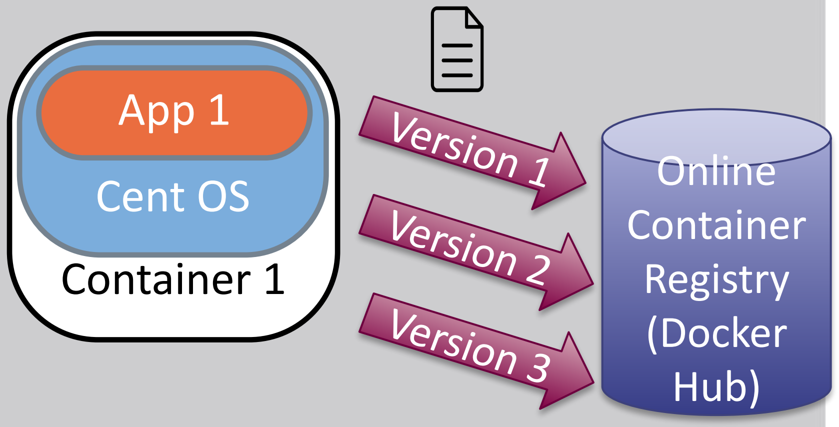
Cons

- Microservices orchestration can be complex (Mesosphere, Kubernetes etc.).

Docker Microservices Lifecycle



Reproducibility with Containers



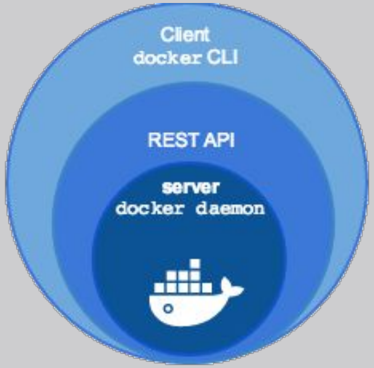
www.docker.com/what-docker

Exact container image

can be used on any other machine, at any desired version.

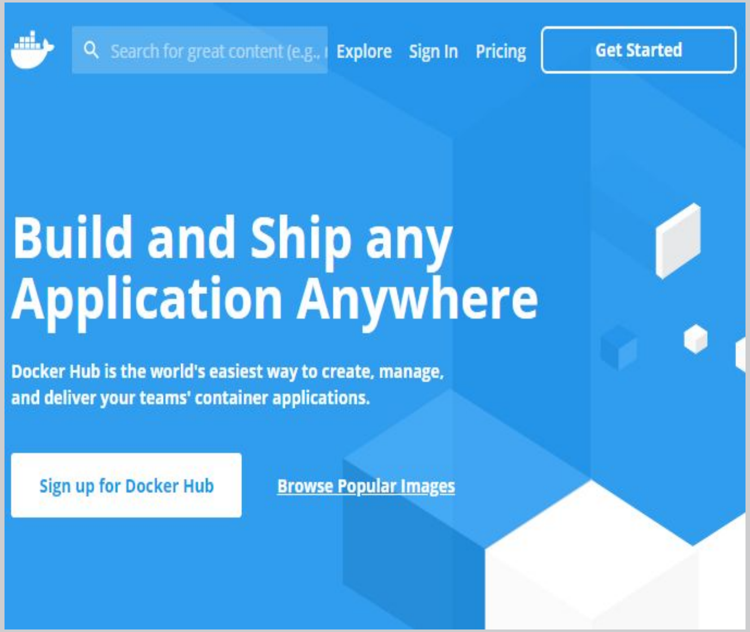
Docker Installation

- Docker Installation Guide:
  - https://docs.docker.com/install/
- Detailed information on how to install docker on different OSs.



Docker Hub

- Make sure you create an account and save your username and password.
- “Docker Hub is the world's easiest way to create, manage, and deliver your teams' container applications”.
- https://hub.docker.com/



Build and Ship any Application Anywhere

Docker Hub is the world's easiest way to create, manage, and deliver your teams' container applications.

Poll: Your Operating System

Which OS are you planning to use Docker on:

- ❏ MS Windows
- ❏ Linux/Unix
- ❏ Mac
- ❏ Other

Docker Installation - Windows 10 Pro

- Docker for windows can be downloaded from: [Docker on Windows]
- System Requirements:
  - Windows 10 64-bit: Pro, Enterprise, or Education (Build 15063 or later).
  - Hyper-V and Containers Windows features must be enabled.
  - Get started with Docker for Windows

My demo: https://www.youtube.com/watch?v=9WqI_6ZFIZs

Docker Installation - Mac

- Docker for Mac can be downloaded from: Docker Desktop on Mac
- System Requirements:
  - Mac hardware must be a 2010 or newer model, with Intel’s hardware support for memory management unit (MMU) virtualization, including Extended Page Tables (EPT) and Unrestricted Mode.
  - macOS must be version 10.13 or newer.
  - At least 4 GB of RAM.

My demo: https://www.youtube.com/watch?v=UQZLfK9UW_c

Docker Installation - Linux

- On Fedora:

```
$ sudo yum install docker-io
```

- On CentOS:

```
$ sudo yum install docker
```

- On Debian and derivatives (i.e. Ubuntu):

```
$ sudo apt-get install docker.io
```

My demo: https://www.youtube.com/watch?v=0Lp3a7rH0ok

End of Part 1

Run a container: *hello-world*

Once you have Docker installed and running .. you can run the hello-world container:

```
$ docker run hello-world
```

if the Docker image is not available on your system, Docker will download (pull) it from your docker registry (default is docker hub).

You will see details in the command-line (terminal).

Run a container: busybox

```
$ docker run busybox echo hello world
```

BusyBox is known as Swiss Army Knife of Embedded Linux.

```
$ docker run -it busybox
```

This will drop you into an sh shell to allow you to do what you want inside a BusyBox system.

Run a container: Ubuntu

```
$ docker run -it ubuntu bash
```

- This is a brand new container.
- It runs a minimal install ubuntu system.
- It will take you inside the container (Ubuntu terminal).
- https://hub.docker.com/_/ubuntu

Run Containers in the Background

- We have seen some examples of how to run containers .. but they were for short jobs .. i.e. foreground containers.
- Long running jobs should/can be run in the background.
- To run a container in the background, we use the -d option:
  - *$ docker run -d -p 80:80 --name=mynginx nginx*

Nginx is a web server which can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache.

- Get Logs: *$ docker logs mynginx*
- Start Container: *$ docker start mynginx*
- Stop Container: *$ docker stop mynginx*
- Delete Container: *$ docker rm mynginx*

The IP Address of a Container

We can find the IP address of a container in the information we retrieve by using the *docker inspect* command:

```
$docker inspect -f '{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
container_name_or_id
```

Searching for Images

- A large number of images exist (keeps increasing).
- There is a good chance that what we want is already there.
- So how do we check?
- The command is as easy as:
    - *$ docker search nginx*
- The output has useful information:
    - “Stars” tell us how popular the image is.
    - Images in the root namespace are “Official”.
    - If images are built automatically by Docker Hub they are “Automated” .. This means that their build recipe is always available.
    - More on images soon!

Docker Images

Here we will learn:

- What is an image.
- What is a layer.
- The various image namespaces.
- How to search and download images.

What is a Docker Image?

- An image is a collection of files + some meta data (Technically: those files form the root filesystem of a container).
- Images are made of layers, conceptually stacked on top of each other.
- Each layer can add, change, and remove files.
- Images can share layers to optimize disk usage, transfer times, and memory use.

Object-oriented Programming (OOP)

- Images are conceptually similar to classes.
- Layers are conceptually similar to inheritance.
- Containers are conceptually similar to instances.

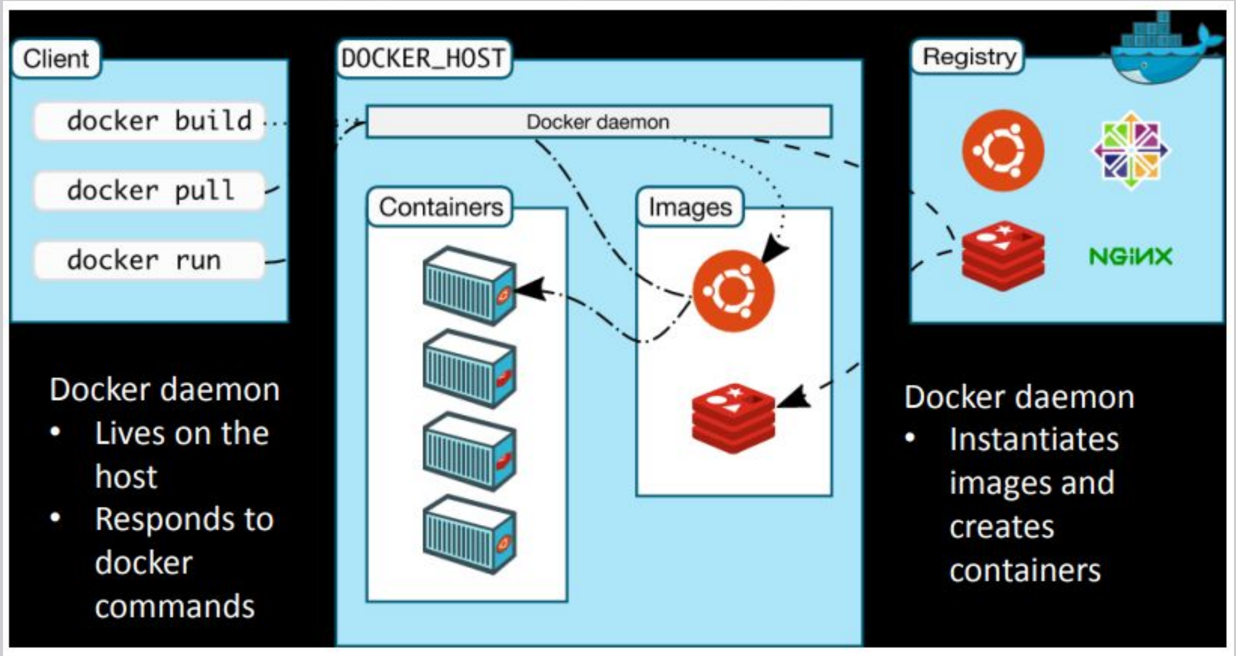
Image Layers

- A Docker container image is structured in terms of layers
- Process for building image:
    - Start with base image.
    - Load software desired.
    - Commit base image+software to form new image.
    - New image can then be base for more software.
- When an image is updated, only new layers are updated.
- Unchanged layers do not need to be updated.
- Consequently, less software is transferred and an update is faster.

Docker Image vs Container

- An image is a read-only filesystem.
- A container is an encapsulated set of processes running in a read-write copy of that filesystem.
- To optimize container boot time, copy-on-write is used instead of regular copy.
- *docker run* starts a container from a given image.

Docker Architecture .. revisited



Docker daemon

- Lives on the host
- Responds to docker commands

Docker daemon

- Instantiates images and creates containers

A container is formed by instantiating an image

Image is Read-Only?

How do we change it?

We do not ..

- We create a new container from that image.
- Then we make changes to that container.
- When we are satisfied with those changes, we transform them into a new layer.
- A new image is created by stacking the new layer on top of the old image.

Practically .. How to Create an Image

- There are multiple ways to create new images.
- We will look at two ways:
  - *docker commit*: creates a new layer (and a new image) from a container.
  - *docker build*: performs a repeatable build sequence.

How to Store and Manage Images

Images can be stored:

- On your Docker host.
- In a Docker registry (remember Docker Hub?).

You can use the Docker client to download (pull) or upload (push) images.

In other words: you can use the Docker client to tell a Docker server to push and pull images to and from a registry.

Root Namespace

- Docker Inc. keeps the root namespace for official images.
- These images are generally authored and maintained by third parties.
- Those images include:
  - Small, “swiss-army-knife” images like busybox.
  - Distro images to be used as bases for your builds, like ubuntu, fedora ...
  - Ready-to-use components and services, like redis, postgresql …

User Namespace

- When you create a Docker Hub account it becomes your namespace.
- It holds your docker images.
- For example:
  - nsadawi/linreg.
  - The Docker Hub user is: nsadawi.
  - The image name is: linreg.
  - https://hub.docker.com/r/nsadawi/linreg

Self-Hosted Namespace

- It is possible to create your own Docker registry to host Docker images.
- Normally you need to publish the hostname (or IP address), and optionally the port, of your registry's server.
- For example:
  - localhost:5000/wordpress
  - The remote host and port is: localhost:5000.
  - The image name is: wordpress.

Example: PhenoMeNal public docker registry

Building Images Interactively 1/4

Building an image interactively is easy as follows:

- Run a container.
- Update the container (e.g. install/uninstall tools).
- Commit the changes manually with docker commit.

Demo: Let us run a docker container, update it and then create an image.

Feel free to use debian, centos, or fedora if they are your favourite choice

Building Images Interactively 2/4

- Start an Ubuntu container (and go inside it):

  $ docker run -it ubuntu bash
- Update Ubuntu: *$ apt-get update*
- Install wget tool (a free tool to download files and crawl websites via the command line): *$ apt-get install -y wget*
- Exit the container: *$ exit*
- Check the changes: *$ docker diff <containerID>*

  The docker diff command lets us see the difference(s) between the base image and our container.

Building Images Interactively 3/4

Docker tracks filesystem changes

We have learned that:

- An image is read-only.
- When we make changes, they happen in a copy of the image.
- Docker can show the difference between the image, and its copy.
- For performance, Docker uses copy-on-write systems. (i.e. starting a container based on a big image doesn't incur a huge copy.).

Building Images Interactively 4/4

- Now let us commit and run the image.
- The *docker commit* command will create new layer(s) with those changes, and a new image using this new layer:

```
$ docker commit <containerID>
```

- The output of the docker commit command will be the ID for your newly created image.
- We can run a container from this image to make sure it has the tools we expect:

```
$ docker run -it <imageID> bash
```

Tagging Images

- Using IDs to deal with images is not practical.
- Better if we can give them names (or tags).
- The docker tag command helps us do just that:

```
$ docker tag <imageID> imageName
```

- In fact we can also specify the tag as an extra argument to commit:

```
$ docker commit <containerID> imageName
```

- We can run using imageName:

```
$ docker run -it imageName bash
```

Useful Docker Commands

- Run container interactively (in fg): $ *docker run -it imageName bash*
- Run container in bg: $ *docker run -d imageName*
- List currently running containers: $ *docker ps*
- List all containers: $ *docker ps -a*
  - to see IDs only use -aq
- Stop a container: $ *docker stop <containerID> OR <containerName>*
- Start a container: $ *docker start <containerID> OR <containerName>*
- Remove a stopped container: $ *docker rm <containerID> OR <containerName>*
- Kill (stop & remove) a container: $ *docker kill <containerID> OR <containerName>*
- Check container logs: *$ docker logs <containerID> OR <containerName>*

Useful Docker Commands

- Download an image: $ *docker pull imageName*
- Tag an image: $ *docker tag <imageID> imageName*
- Search for an image: $ *docker search imageName*
- List images: $ *docker images*
  - add *-a* to list even unnamed/untagged images
  - add *-q* to see IDs only
- Delete an image: $ *docker rmi <imageName> OR <imageID>*
- Delete all image: $ *docker rmi $(docker images -a -q)*
- Login to Docker Hub (will ask for credentials): *$ docker login*
- Login to Specific Registry : *$ docker login registryURL*
- Logout: *$ docker logout*
- Remove all unused objects: *$ docker system prune*
- Get information about Docker: *$ docker info*

End of Part 2

Building Images Automatically

- We have seen how to manually build an image (docker commit).
- Now we will learn how to automate the build process by writing a Dockerfile.
- A Dockerfile is a text document that contains all the commands we could call on the command line to assemble an image.
- Using *$ docker build* we can create an automated build that executes several command-line instructions in succession.
- Dockerfile reference

Our First Dockerfile

- Create a directory called MyFirstDockerfile and CD into it.
- Create an empty file called **Dockerfile** (notice no extension).
- Type the following lines into this file and save it (seen them before?):

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
```

- FROM indicates the base image for our build.
- Each RUN line will be executed by Docker during the build (at build time).
- RUN executes command(s) in a new layer and creates a new image.
- It is mainly used for installing software packages.
- Our RUN commands must be non-interactive (No input can be provided to Docker during the build hence the **-y** option).

Build Image from Dockerfile

- Our example Dockerfile uses the official Ubuntu base image.
- It updates Ubuntu and installs the “wget” tool.
- In the terminal issue this command:

```
$ docker build -t myubuntu .
```

- *-t* to tag the image (new image will be called myubuntu).
- You can tag with your Docker Hub ID: *-t nsadawi/myubuntu*
- The *.* indicates the location of the build context (it is possible to have Dockerfile in a different directory).
- Observe the output.
- Let us go through the output.
- Now the image can be run .. the wget tool should be there:

```
$ docker run -it myubuntu bash
```

Viewing Image History

- The history command lists all the layers composing an image.
- For each layer, it shows its creation time, size, and creation command.
- When an image was built with a Dockerfile, each layer corresponds to a line of the Dockerfile.
- In the terminal issue this command:

```
$ docker history myubuntu
```

- Observe the output and try to make sense of it.

CMD and ENTRYPOINT

- The CMD and ENTRYPOINT are essential to know.
- We need to be familiar with them as they give us control on what happens when the container runs.
- They give us the ability to set the default command to run in a container (i.e. what to run as soon as the container starts).
- CMD sets default command and/or parameters, <u>which can be overridden from command line</u> when docker container runs.
- ENTRYPOINT configures a container that <u>will run as an executable.</u>

CMD

- CMD instruction allows you to set a *default* command.
- Command will be executed only when you run container without specifying a command.
- If Docker container runs with a command, the default command will be ignored.
- It can appear at any point in the file.
- If Dockerfile has more than one CMD instruction, all but last CMD instructions are ignored:
  - Each CMD will replace and override the previous one.
  - As a result, while you can have multiple CMD lines, it is useless.

CMD Example

- CMD happens at run-time, RUN happens at build-time
- Dockerfile:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
CMD wget -O- -q http://ifconfig.me/ip
```

- Build and test the image:
  - `$ docker build -t myubuntuip .`
  - `$ docker run myubuntuip`

ENTRYPOINT

- ENTRYPOINT instruction allows us to configure a container that will run as an executable.
- It looks similar to CMD, because it also allows us to specify a command with parameters.
- The difference is ENTRYPOINT command and parameters are not ignored when Docker container runs with command line parameters.
    - There is a way to override it though!
- As with CMD, ENTRYPOINT can receive parameters.

ENTRYPOINT Example

- Dockerfile

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
ENTRYPOINT ["wget", "-O-", "-q"]
```

- Notice the square brackets (exec form, preferred).
- Build and test the image (notice we pass the URL):
  - `$ docker build -t myubuntuip .`
  - `$ docker run myubuntuip http://ifconfig.me/ip`

RUN, CMD and ENTRYPOINT

- Use RUN instructions to build your image by adding layers on top of initial image.
- Prefer ENTRYPOINT to CMD when building executable Docker image and you need a command always to be executed.
- Choose CMD if you need to provide a default command and/or arguments that can be overwritten from command line when docker container runs.

Docker Volumes

- The stuff we have seen so far does not help us store any data from the container permanently.
- In order to store data from the container into our host machine we need to use Docker Volumes: Use volumes
- To mount a host volume while launching a Docker container, we have to use the following format for volume **-v**:
  - *-v HostFolder:ContainerVolumeName*
- Normally done in the command line .. NOT in the Dockerfile.

Mounting Volumes inside Containers

- The -v flag mounts a directory from your host into your Docker container.
- The flag structure is:
  - [host-path]:[container-path]:[rw|ro]
- If [host-path] or [container-path] does not exist it is created.
- You can control the write status of the volume with the ro and rw options.
- If you do not specify rw or ro, it will be rw by default.

Docker Volume Example

- For example, if we run the busybox container: *$ docker run -it --name=container1 -v /home/noureddin:/datavol busybox*
- What we have done here is: we have mapped the host folder /home/noureddin to a volume /datavol that will be mounted inside our container (container1).
- The datavol folder in the container should have the contents of the /home/noureddin on the host (any changes will be reflected).
- Let us try it!

Putting it all together!

- This example will involve what we have learned so far.
- Let us create an image that:
  - is based on Ubuntu 20.04 (It has Python 3 already installed).
  - We will update Ubuntu and install the sklearn, pandas packages on it.
  - Write a Python script that receives a CSV file, perform some calculations on it and write the results into a results file.
  - This results file should be saved into our host machine (permanent).

Our Dockerfile

```
# from Ubuntu 20.04 base image
FROM ubuntu:20.04
# provide your contact info if you wish
MAINTAINER Noureddin Sadawi, myemail@mail.com
# run these commands ..
RUN apt update
RUN apt install -y python3-pip
RUN pip3 install pandas scikit_learn
# copy this Python script from host machine to docker image
ADD linear-regression.py /
# as soon as a container starts, run this script using Python3
ENTRYPOINT ["python3", "/linear-regression.py"]
```

Build Image and Run Container

- Let’s build our image (you can choose your own tag):
  - `$ docker build -t linreg .`
- and now we run the container passing it our dataset file and specifying the folder we want to the result to be stored in (using volumes):
  - `$ docker run -v /path/to/dockerfile/:/work/ linreg dataset.csv`
- This *`/path/to/dockerfile/`* is the full path to your current directory on the host machine.
- This *`/work/`* is the directory inside the container (the script will write the results there).
- the *`dataset.csv`* file will be automatically passed to the script as an ENTRYPOINT argument.

Run Linux Containers on Windows

- It is possible to run Linux containers on Windows.
- You need to install WSL on Windows.
- More info here: https://docs.docker.com/desktop/windows/wsl/
- **Prerequisites:** Before you install the Docker Desktop WSL 2 backend, you must complete the following steps:

1. Install Windows 10, version 1903 or higher or Windows 11.
2. Enable WSL 2 feature on Windows. For detailed instructions, refer to the Microsoft documentation.
3. Download and install the Linux kernel update package.

Dockerfile for Windows IIS Server

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019
RUN powershell -Command Add-WindowsFeature Web-Server
CMD [ "ping", "localhost", "-t" ]
```

- FROM mcr.microsoft.com/windows/servercore:ltsc2019, tells Docker to use the image as a base to build this new image.
- The RUN line is telling Docker to run some PowerShell commands that turn on the Web-Server Windows feature.
- CMD is just a persistent ping to give the container something to hang off of for demonstration purposes.

Build Image and Run Container

- Let’s build our image (you can choose your own tag):
  - *$ docker build -t myiis .*
- That will take a few minutes (Windows images/containers are big).
- and now we run the container passing it our dataset file and specifying the folder we want to the result to be stored in (using volumes):
  - *$ docker run -p 8080:80 myiis*
- Open a Web Browser and navigate to localhost:8080
- You will see the IIS main page .. it is from the container.
- You can add your own content to the website using the COPY command.

End of Part 3

thank you
рахмат
danke
謝謝
ngiyabonga
teşekkür ederim
Баярлалаа
спасибо
faafetai lava
mersi
kia ora
barka
welalin
tack
spas
vinaka
спасибі
blagodaram
dank je
misaotra
matondo
paldies
grazzi
mahalo
tapadh leat
gracias
kiitos
dankie
dhanyavad
nanni
nandri
mauruuru
hvala
köszönöm
akun
dankon
ačiū
хвала
asante
manana
obrigada
murakoze
tenki
bedankt
bayarlalaa
grąçie
enkosi
dziękuję
djiere dieuf
tau
mochchakkeram
дякую
mamnun
chokrane
chnorakaloutioun
gratias ago
gràcies
sulpáy
go raibh maith agat
sobodi
dĕkuji
sagolun
sukriya
kop khun krap
taiku
obrigado
mèsi
didi madloba
ありがとう
najis tuke
tanemirt
grazie
arigatô
takk
dakujem
trugarez
kam sah hamnida
rahmat
terima kasih
rahmet
diolch
dhanyavadagalu
shukriya
mercê
мерси
তোমাকে ধন্যবাদ
감사합니다
xièxie
ευχαριστώ
merci