



An introduction to Docker

github.com/defano/docker-tutorial

Matt DeFano
IOT TECHNOLOGY SOLUTIONS

Part 1: An introduction to Docker (approx 40 min)

- ✓ What is Docker? What's it good for?
- ✓ Docker key concepts and terminology
- ✓ A brief command-line reference
- ✓ Other products in the ecosystem

Part 2: Hands-on lab exercises: (as time allows)

- ✓ Pull images from Docker Hub and run them
- ✓ Create your web server image
- ✓ Link it to a Redis cache container
- ✓ Orchestrate the system with Compose

Can't follow along tonight? Step-by-step lab instructions are available on Github:
github.com/defano/docker-tutorial



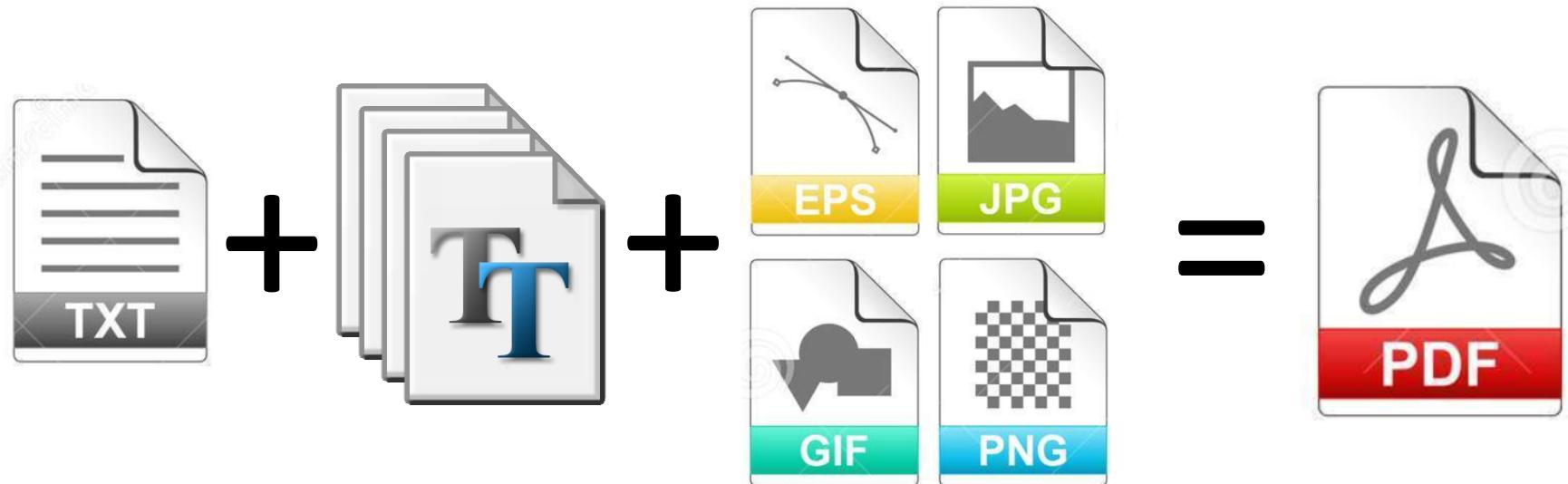
Docker lets you package an application with all its dependencies and infrastructure and execute it anywhere the Docker Engine is installed.

“Build, ship, run”



Docker Engine ... for building and running Docker Containers	Docker Swarm ... for clustering Docker hosts
Docker Compose ... for provisioning multi-container Docker applications	Docker Hub Registry ... for publishing Docker images to the community
Docker Trusted Registry ... for organizations that need a private Docker registry	Docker Cloud ... for managing Docker hosts on a cloud service provider (like AWS)
Docker Machine ... for running Docker on machines that aren't natively supported	Universal Control Plane ... for IT departments to deploy and secure Docker apps in their firewall
Docker Toolbox & Kitematic ... for getting started quickly on a Mac or Windows machine	Docker Notary ... for signing Docker images and managing trust relationships

Red denotes tools we'll discuss in this presentation.

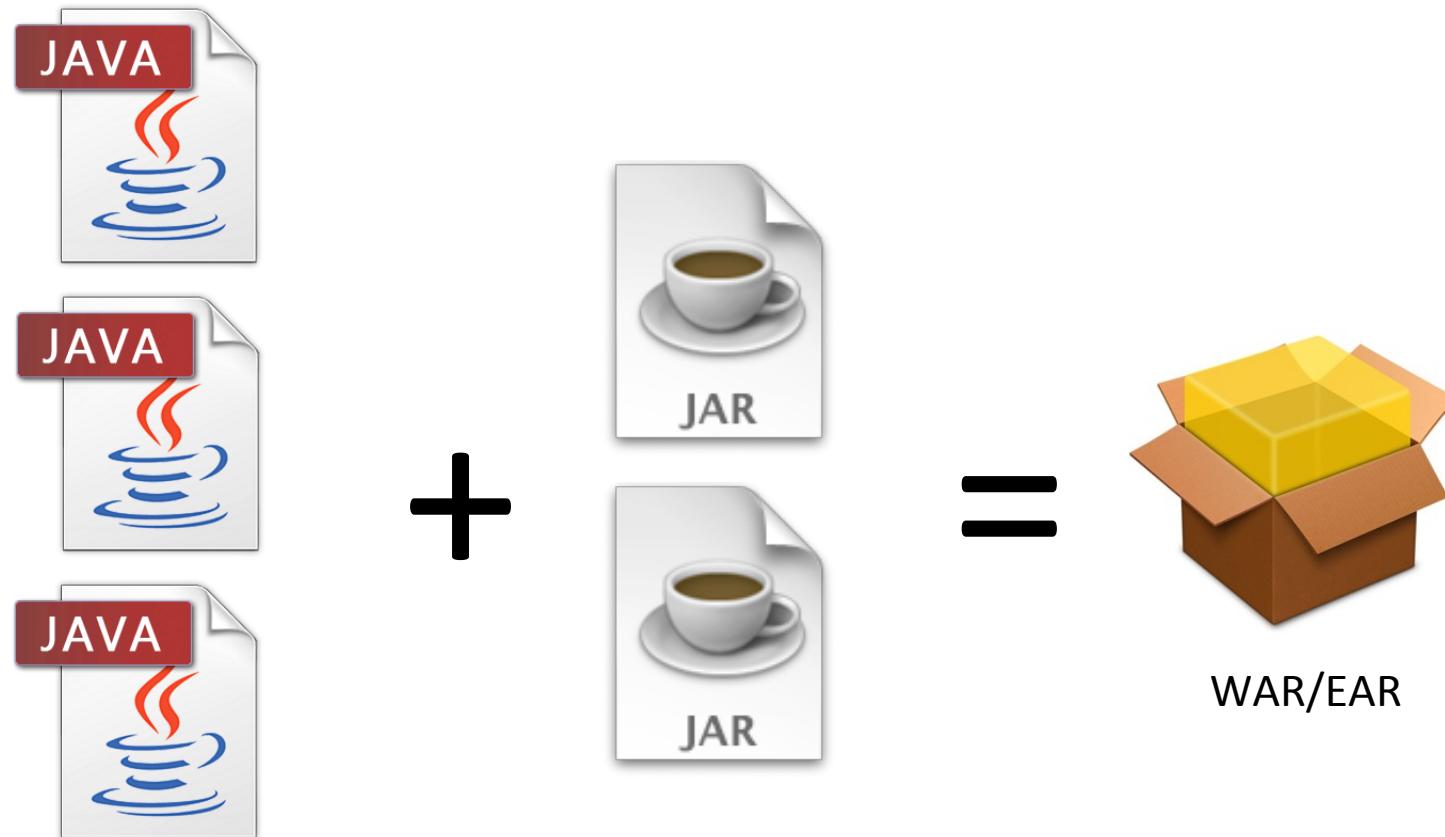


Your opinions on
Shakespeare...

... in a bunch of fonts
you bought at
Babbage's...

... decorated with clip
art from The Print
Shop, looks like...



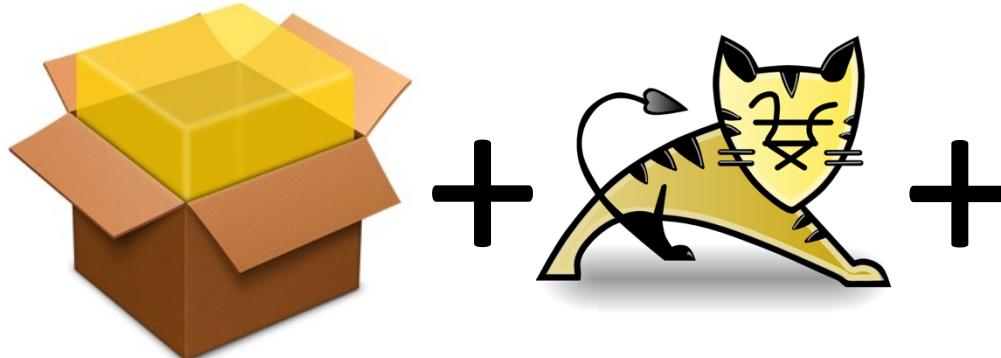


Your code...

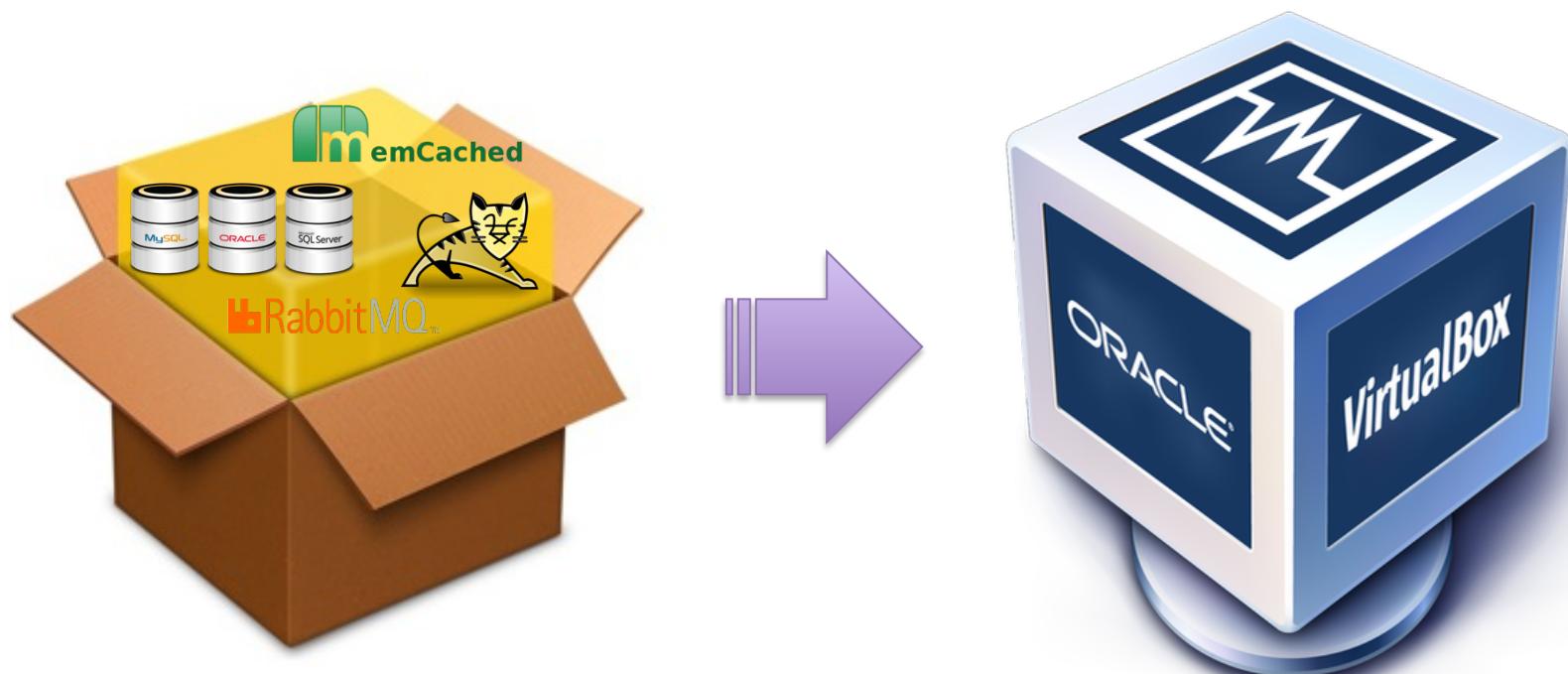
... plus a bunch of third
party libraries...

... equals an industry-standard,
run anywhere package!

Sure, until this happens...



In that case, I'll use a VM

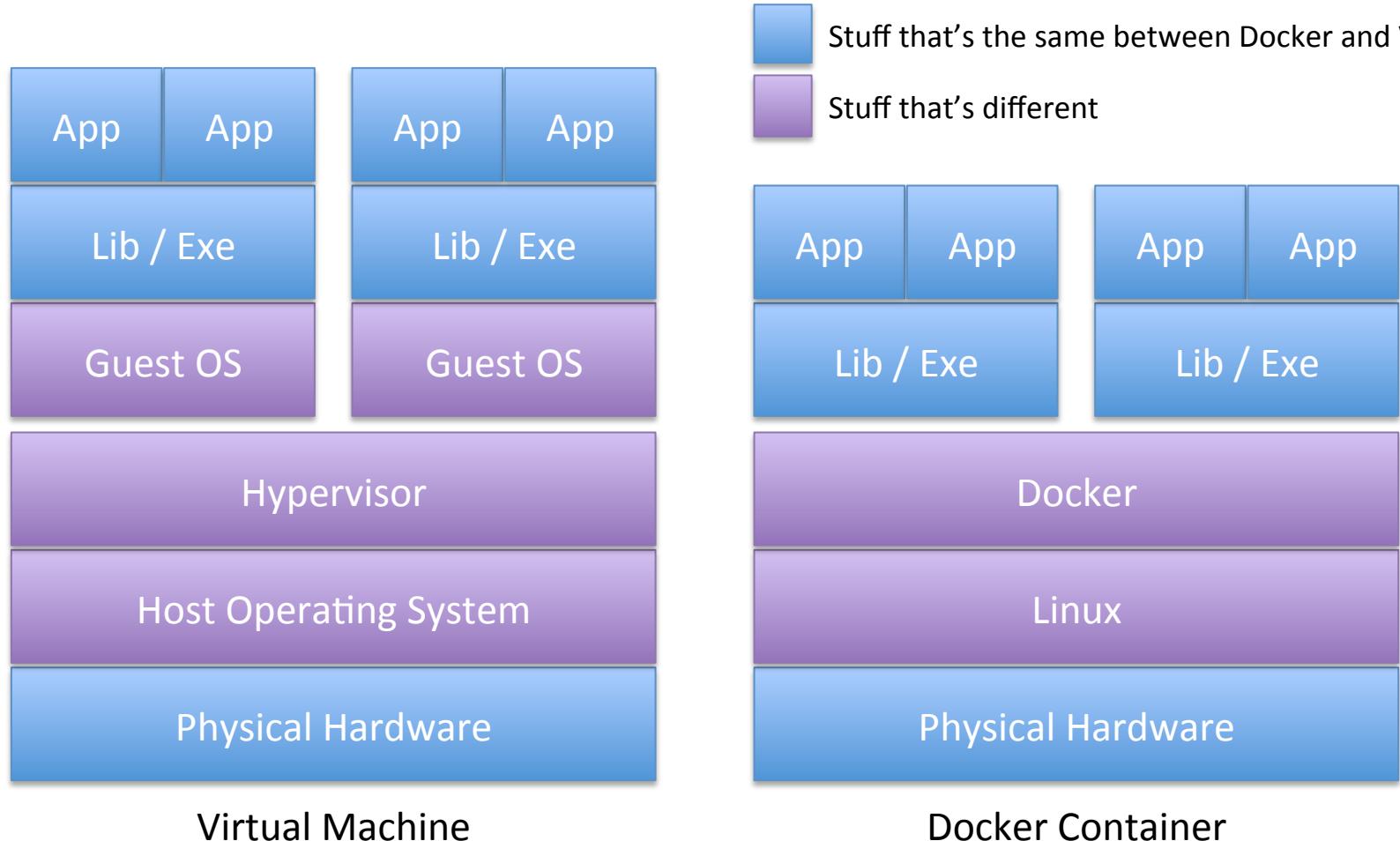


Virtual Machines:

- ✓ Virtualize physical hardware; VMs share host hardware
- ✓ Start in 30-60 seconds
- ✓ Make cloud-to-cloud portability hard
- ✓ Machine images are big and atomic
- ✓ Cryptacular for development
- ✓ More secure
- ✓ Better for environments with lots of OSs; many apps running on one machine

Docker Containers:

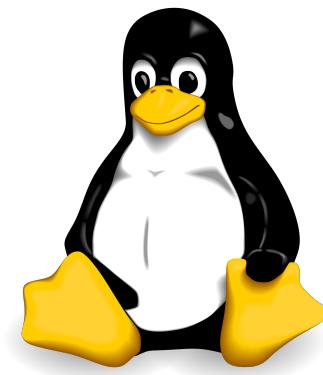
- ✓ Virtualize Linux; containers share hardware *and* OS.
- ✓ Start in 0.5 seconds
- ✓ Make cloud-to-cloud portability easy
- ✓ Container images are composite and reusable
- ✓ Great for development
- ✓ Less secure
- ✓ Better for Linux-only environments and micro-services



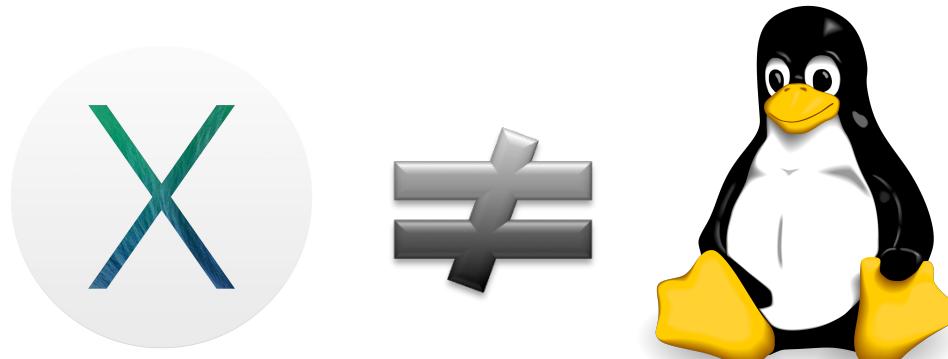
**Illinois state law requires that every presentation about Docker show this diagram.*

Repeat after me...

Containers *always* run on Linux.



Cool. So containers run on my Mac, right?



Nope. Docker is reliant on kernel features not present in Apple's Darwin/XNU kernel.

No Linux, no problem: Run containers with **Docker Machine**.

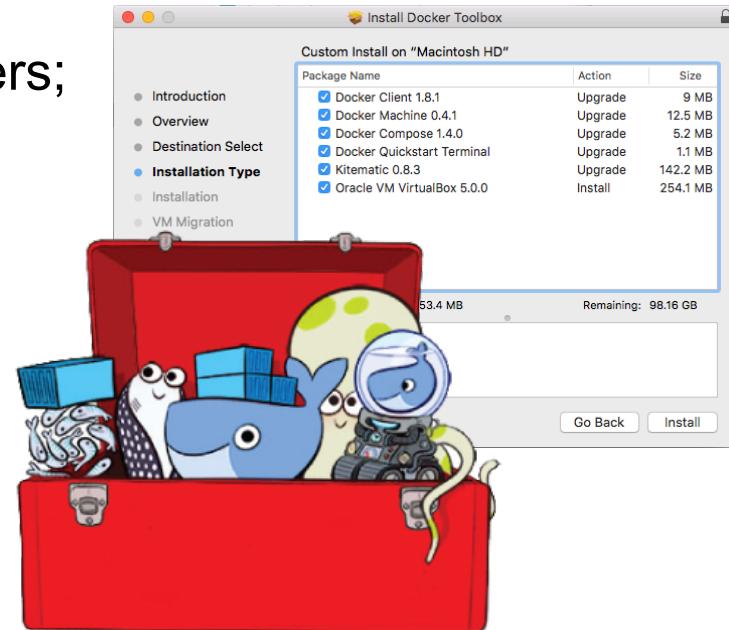
- ✓ Typically just Linux running in a virtual machine.
- ✓ Mac and Windows docker command knows how to tunnel into your Docker Machine (makes it look like Docker is running on your computer).
- ✓ Docker Machine supports drivers for different VM engines and cloud providers; most developers just use VirtualBox.



Mac and Windows users can get started quickly with the **Docker Toolbox**

One-stop shop for Mac and Windows users; installs everything you'll need:

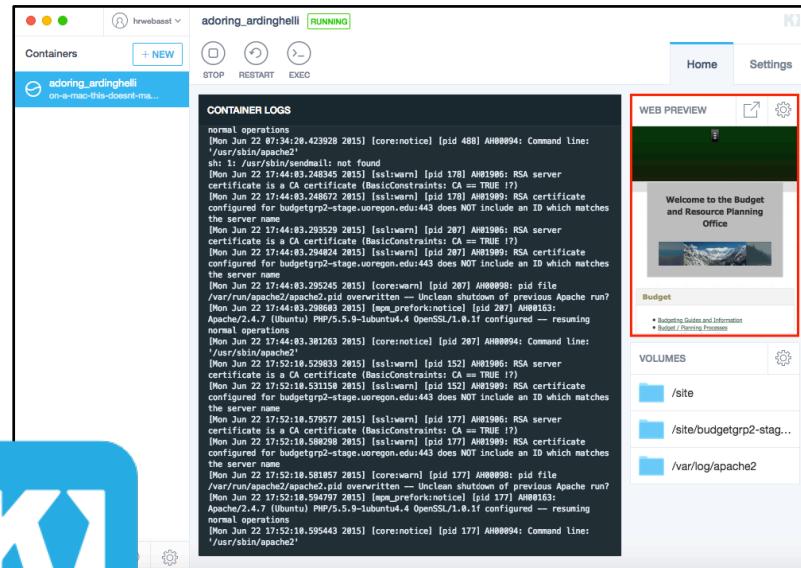
- ✓ Docker Machine
- ✓ Docker Engine
- ✓ Docker Compose
- ✓ Docker Kitematic
- ✓ Oracle VirtualBox



Linux users can install these tools individually.

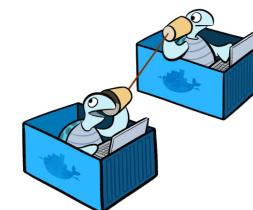
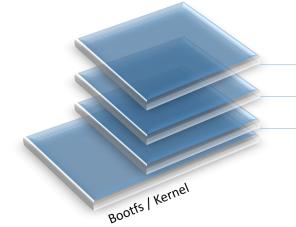
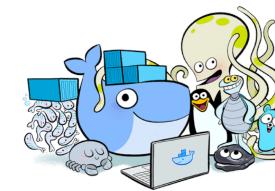
Kitematic is an easy-to-use Docker GUI for Mac and Windows.

- ✓ Browse and create containers from Docker Hub directly within the app.
- ✓ Provision containers, view logs, map ports and preview websites.
- ✓ No need to learn command-line syntax.



Master Docker by understanding these **key concepts**:

- ✓ Understand the difference between **images**, **containers** and **machines**.
- ✓ Know how images are constructed through a **union mount**.
- ✓ Don't let the **port-spaghetti** of containers running inside virtual machines running on your laptop make your head explode.



A **container** is an instance of a Docker image.



An **image** defines the contents of the filesystem, plus some defaults:

- Exposed network ports
- Volumes (writeable disk space on the host)
- Linux shell commands to initialize container (i.e., start your app)



Images and containers are identified by **ID** or **name** and **tag**.

- **ID** is a hash assigned by Docker
- **Name** is human-readable value:
 - ✓ Optional for images
 - ✓ Auto-assigned to containers when not specified (i.e., `nice_schnauzer`)
- **Tag** is a human-readable value:
 - ✓ For images only (no tags on containers); typically used to denote version
 - ✓ When not explicitly specified, `latest` is used.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
webserver	1.0	3b528a2de778	28 minutes ago	227.7 MB

A **volume** is a directory on the Docker host that's read / writeable to the container.

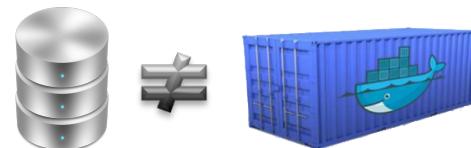


A container communicates to the outside world through one or more **exposed network ports**.



Images are not monolithic.

- ✓ A Docker image *is not* a simple “disk image”



Images are comprised of other images, one atop another, merged together in a **union filesystem**.

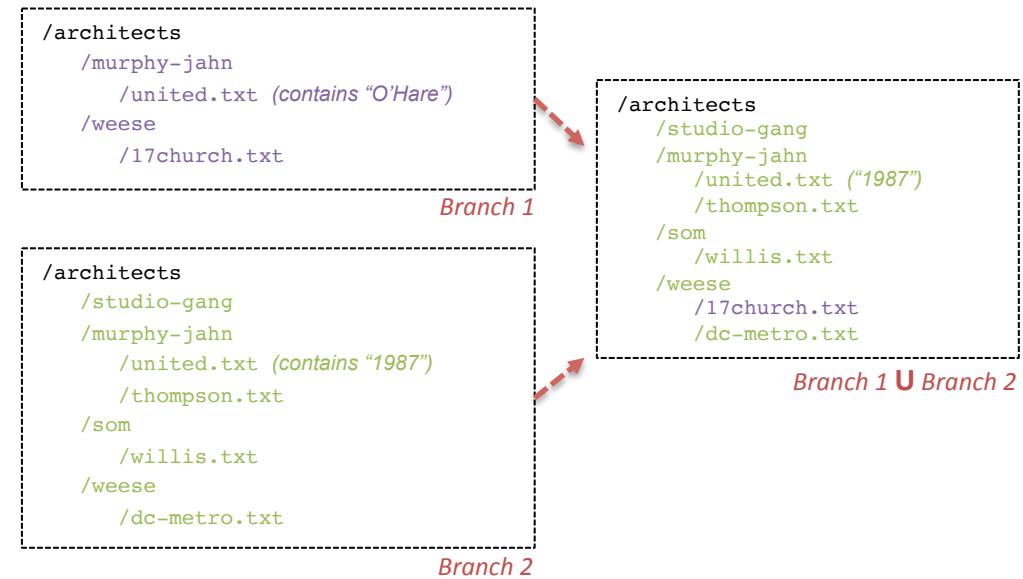
- ✓ Docker manages these relationships for you:

A reference to an image automatically includes all its dependent layers.



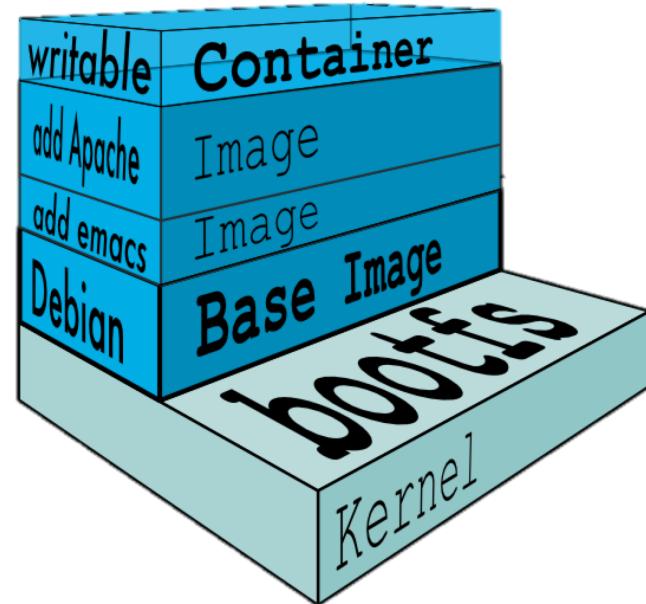
Okay, but what the heck is a **union filesystem**?

- Mounts multiple filesystems (called “**branches**”) at the same mount point.
- Non-conflicting nodes appear merged together in union: $n_1 \cup n_2 \cup n_3$
- Conflicting nodes appear in mount order (last mounted node wins)
- **Read only**. What happens when you write?



Which layer is modified **when I write** to Docker's union filesystem?

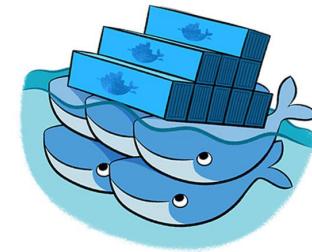
- The top-most layer is provided by the container and is writeable.
- This writable layer is discarded when container stops running. **Your changes are not saved.**
- Writable layer can be committed to create a new Docker image. (As a matter of fact, this is what's happening when the Dockerfile creates new images.)



Daddy, where do **Docker images** come from¹?

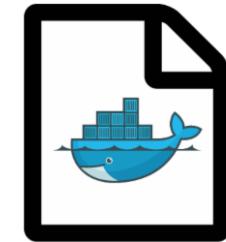
- **Pull them** from the Docker Hub

The open source community has “Docekerized” virtually everything you could imagine.



- **Create them** from a Dockerfile

Like a makefile for Docker; lets users script a series of instructions for building an image.



- **Both.** At the same time.

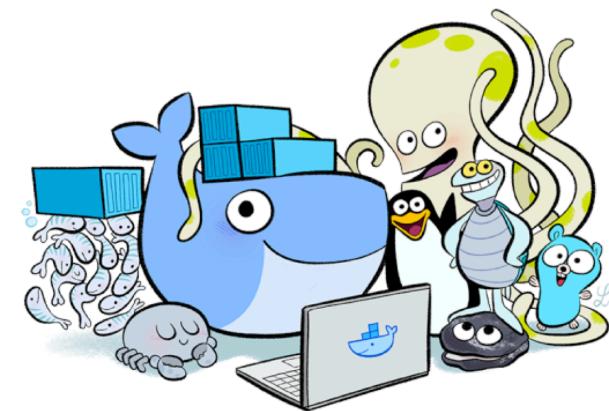
Most homemade images start from a base image on the Docker Hub Registry.



¹ From the Docker Stork, of course.

Docker Hub Registry is a repository for open-source images.

- Search for images on the web at <http://hub.docker.com> or use
`docker search <term>`
 - Image names typically following the form `<organization>/<image>:<tag>`
 - Use tags to denote versions, variants, etc.
- Pull an image (for use on your host) with
`docker pull <image>`
- Share your masterpiece with the world:
`docker push <image>`
 - Worry not; required layers will be pushed, too.



What does a **Dockerfile** look like?

FROM specifies the base image we're starting from. Pulls the "java:8" image from Docker Hub.
Dockerfiles must start with FROM;
Find images at hub.docker.com.

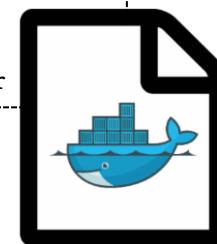
RUN starts an instance of the java:8 image, executes apt-get update inside of it, then "commits" the changes made to the filesystem to a new image.

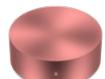
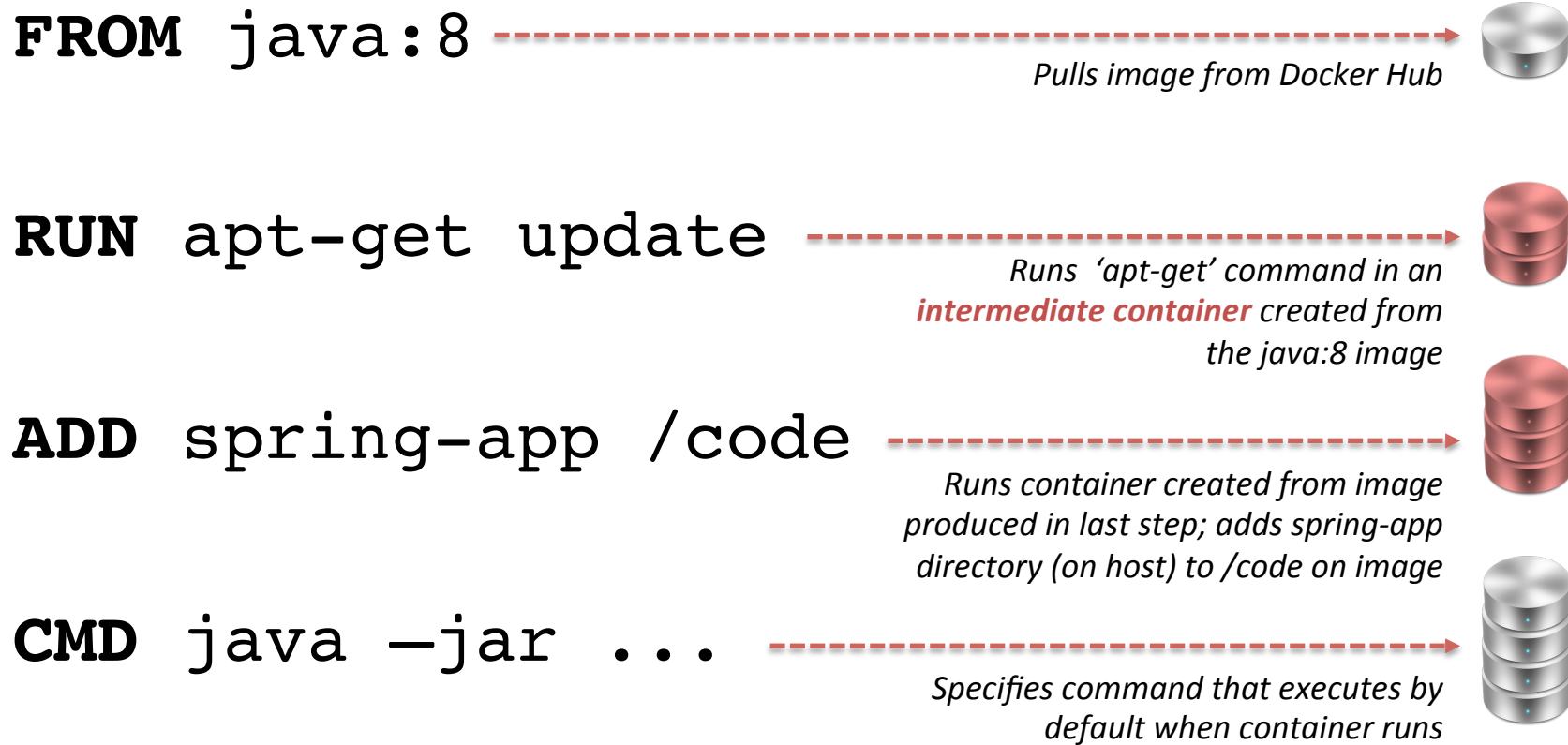
ADD copies the spring-app directory into the container and commits the change as a new image.

EXPOSE specifies ports that are accessible from the host.

```
FROM java:8  
  
RUN apt-get update && apt-get install -y maven  
  
ADD spring-app /code  
RUN mvn package -DskipTests=true  
  
EXPOSE 8080  
  
CMD java -jar target/spring-example-1.0.0.jar
```

CMD defines the command that should be executed when a container created from this image starts.





Intermediate containers are deleted automatically (by default) when build completes.

Create an **image** from a Dockerfile:

(The trailing ‘.’ is the path to the Dockerfile)

```
$ docker build -t webserver:1.0 .
```

Create a named **container** from an image:

(Call the container “defano/server” and publish the container’s exposed port 80 on the Docker Host at 8080)

```
$ docker create -p 8080:80 --name="my-server" webserver:1.0
```

Start an existing **container** by name:

(We could also specify the container by hash prefix)

```
$ docker start my-server
```

Run a command in a new container...

(Creates a new container from the webserver:1.0 image and executes ping inside of it)

```
$ docker run webserver:1.0 /bin/ping www.google.com
```

... or run the image with its default command:

(When no command is specified, the CMD from the Dockerfile will execute)

```
$ docker run webserver:1.0
```

Execute a command in a running container:

(Opens a bash shell in the container named “defano/server”; -it indicates command is interactive and needs a PTTY)

```
$ docker exec -it my-server bash
```

List all Docker containers:

(Forego the `-a` to list only running containers)

```
$ docker ps -a
```

List all Docker images in your local repository:

(Add a `-a` to show intermediate images, too)

```
$ docker images
```

Show logs (output) of a container:

(Use `-f` to “follow” the output; otherwise, display current output and exit. Works on running and stopped containers!)

```
$ docker logs -f my-server
```

Delete a container:

(Deletes the container named “my-server”; could also specify a hash. Container must not be running.)

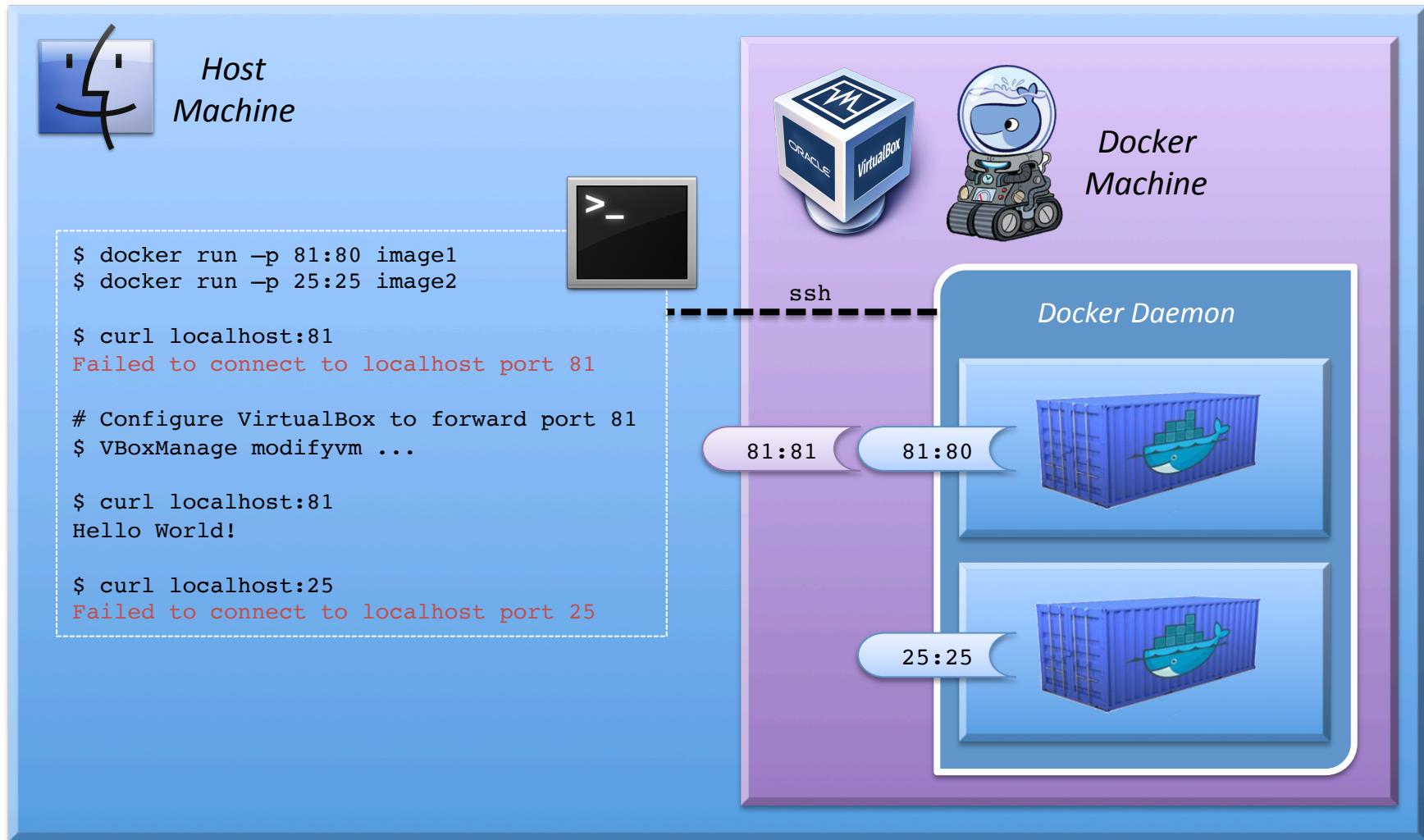
```
$ docker rm my-server
```

Delete an image:

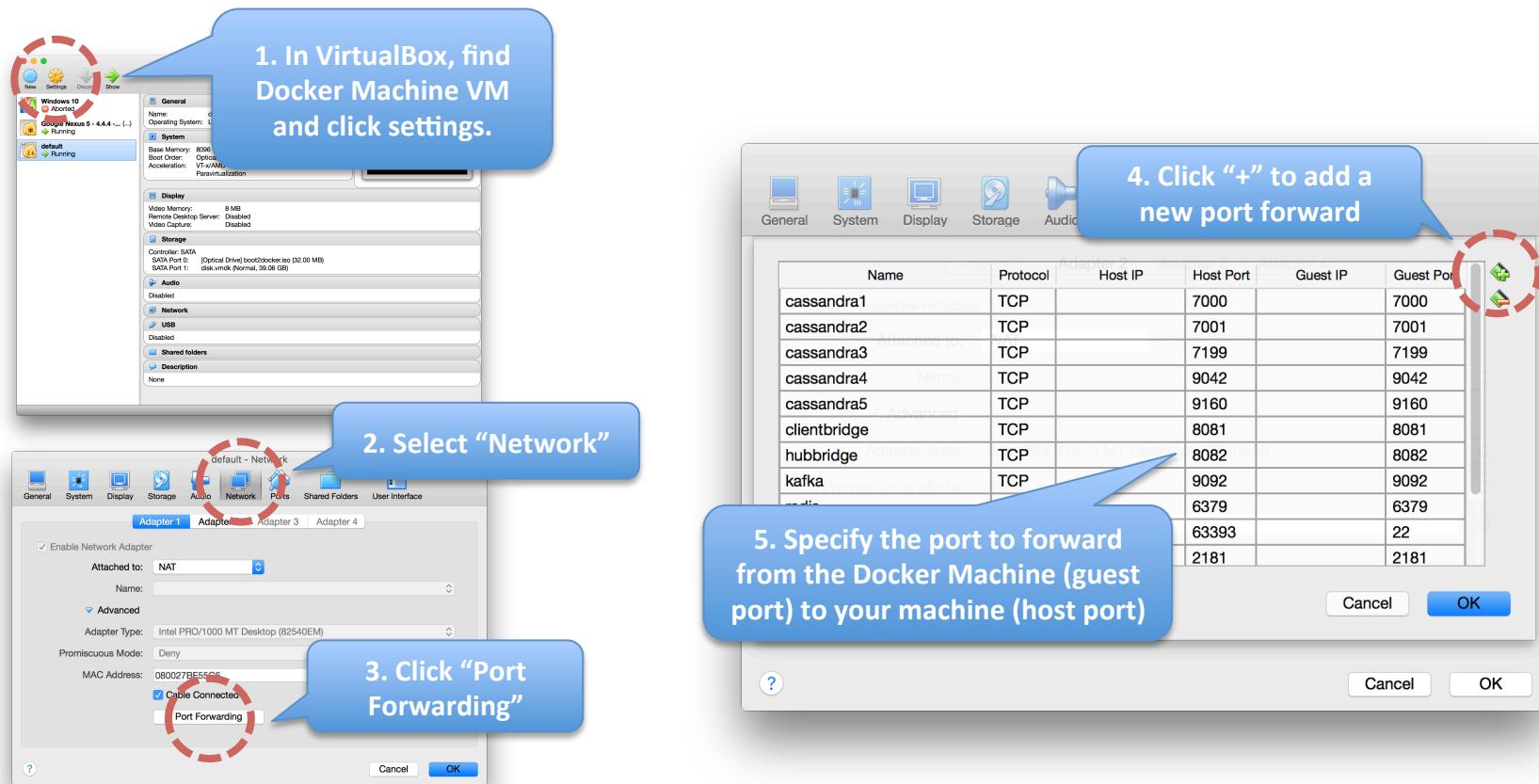
(Deletes the webserver:1.0 image; could also specify a hash. All derived containers must be removed first.)

```
$ docker rmi webserver:1.0
```

Plus many, many more. See the Docker command-line reference for more advanced options: <https://docs.docker.com/engine/reference/commandline/cli/>



How can I **forward ports** to my Mac or Windows?

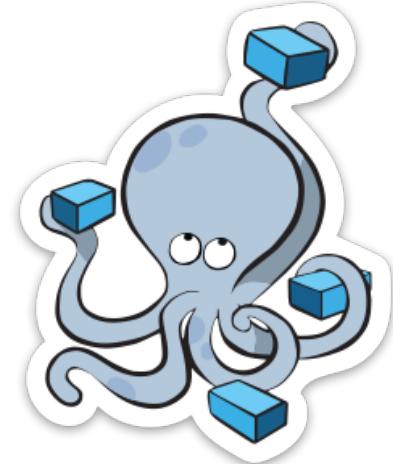


Use **Docker Compose** to script the creation of multi-container applications

- ✓ **Great for development environments**
Starting and linking lots of containers is a pain.

- ✓ **Great for CI system tests**
Start a multi-container app with docker-compose up, then execute your tests.

- ✓ **Great for single-host deployments**
Running all your containers on one physical host, then this is your tool.



Matt DeFano
Sr. Architect, Manager

IOT TECHNOLOGY SOLUTIONS
An STA Group Company

222 South Riverside Drive
Suite 2800
Chicago, IL 60606

matt.defano@iottechnology.com
www.iottechnology.com

