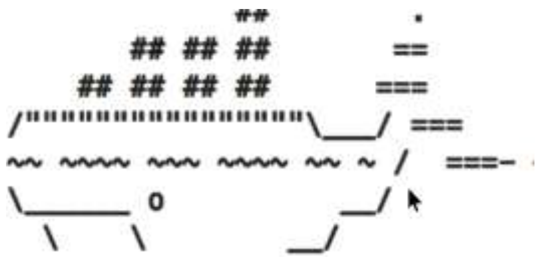
# Practical Tutorial Micro-services & Docker
# Azure CLI and Resource Manager



## Lab 05

By Joan Imrich, Nishava, Inc.

## Deep Azure @McKesson

# Goals of Lab 05

- Practical tips, **Key Concepts & Demos** on Docker Microservices, Azure SQL Database & App (Web) deployments
- Share lessons learned through a series of tutorials using Azure Resource Management Templates

| **In Scope … Make sense of Big Picture** | **Out of Scope … In the weeds** |
|---|---|
| Docker Images & Containers, CLI | Database Migration Techniques |
| Microservices & VMs (Ubuntu, CentOS) | Security, Data Masking, Table Auditing |
| Azure Resource Manager (ARM) , git | Business Continuity & DR, Monitoring |
| Deploy App & SQL database , nginx | Operating System Issues (MAC, Win7) |
| Focus on Lecture handouts, URLS | Specific McKesson Configurations |
| General Guidance on Course | Debugging Code / IDE Solutions |
| Useful commands and tools | Deep Dive Alerting & Docker VM Networking |

Interactive Session – Please Ask Questions!

https://docs.docker.com/engine/userguide/networking/

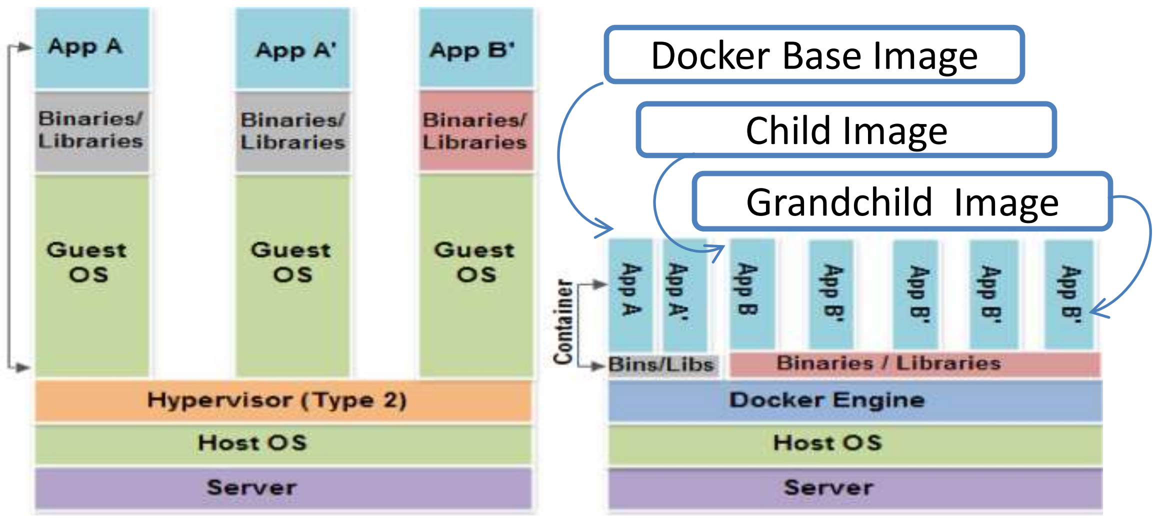
# Virtual Machine Vs Docker Services

**Do you need a full platform that can house multiple services?**

- Go with a virtual machine

**Do you need a single service that can be clustered and deployed at scale?**

- Go with a container



**Containers are to Virtual Machines as threads are to processes**

- Docker containers are much lighter than virtual machines. Concept of layering images allows for amazing speed benefits
- Containers can start in a fraction of time of VMs, different containers are completely isolated but could share libraries, Docker Registry is a hosted service, containers allows greater performance efficiencies

## Docker Microservices - Know the What, Why, & How

### What are the use-case scenarios?

- One of the biggest use-cases for Docker right now is the need to modernize traditional monolithic apps that businesses rely on, but take up a huge amount of the IT budget
- Moving those apps to Docker gives you immediate benefits in portability, efficiency and security - and a roadmap for modernizing the architecture and breaking down the monolith (testing)

### Why invest in microservices? (operational expenses, trade-offs)

- traditional app- basically under-utilized infrastructure (hardware it runs on), and over-utilizing the humans who build and manage IT (reduced budgets)

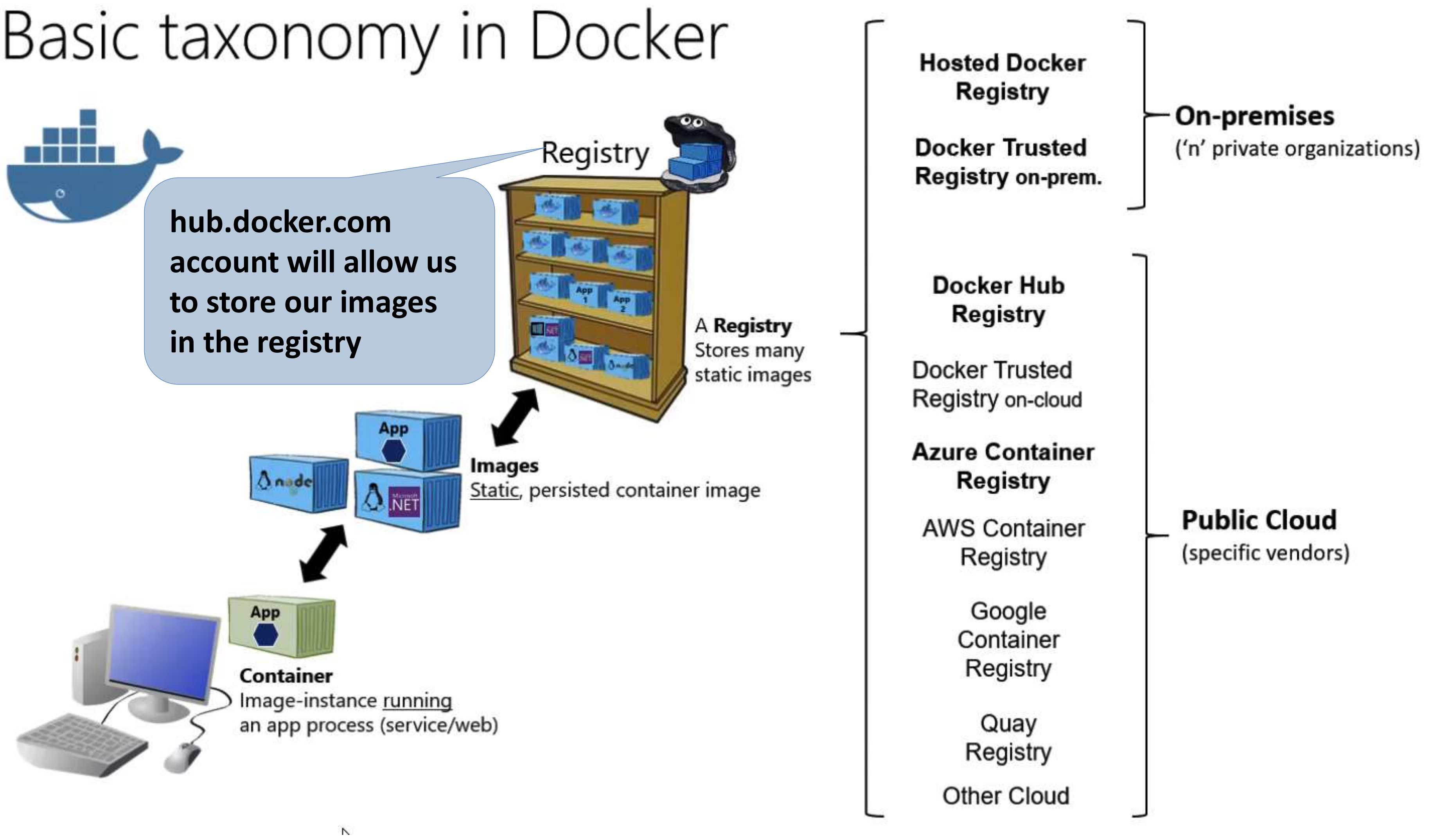
### How does Docker work

- You build Docker images that hold your apps inside Docker containers to run your applications
- You can share those Docker images via Docker Hub or your own registry
- Implemented on **.NET Core** or .NET Framework (depends on cross-platform needs, Win APIs, µS.

**DOCKER OBJECTS & TERMINOLOGY**

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. An ***Image*** is a read-only template with instructions for creating a Docker container. Azure Images (like AWS AMIs) are just templates for docker containers. To build your own image, you create a **Dockerfile** with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. ***Dockerfile*** is a script, composed of various "commands" also called "instructions" and arguments listed successively to automatically perform actions on a ***base image*** in order to create a new one.

# Docker Terms & Concepts


Basic taxonomy in Docker
Registry
hub.docker.com account will allow us to store our images in the registry
A Registry Stores many static images
App
node
.NET
Images
Static, persisted container image
App
Container
Image-instance running an app process (service/web)
Hosted Docker Registry
Docker Trusted Registry on-prem.
On-premises
('n' private organizations)
Docker Hub Registry
Docker Trusted Registry on-cloud
Azure Container Registry
AWS Container Registry
Google Container Registry
Quay Registry
Other Cloud
Public Cloud
(specific vendors)

SOURCE: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/container-docker-introduction/docker-terminology

# Docker Container Microservices

Docker helps developers build & test any app in any language using any toolchain (modular) and ship applications, faster

**BUILD** (Light-Weight)

- Based on Linux **Containers**
- Uses layered filesystem to save space (**AUFS**/LVM)
- Uses a copy-on-write filesystem to track changes

**SHIP** (Self-sufficient)

- A Docker container contains everything it needs to run anywhere (minimal Base OS, libraries, frameworks, app code)

**RUN** (Portable)

- Can run anywhere on most systems (Raspberry pi, Linux distributions, Windows)
- VM vs Container, Cloud Trade-offs depend on use case objectives & goals

## CONTAINER LIFECYCLE

docker create creates a container but does not start it
docker rename allows the container to be renamed
docker run creates and starts a container in one operation
docker rm deletes a container
docker update updates a container's resource limits

- Does a Docker Container have it's own IP private address? Yep. It's like a process with an IP address

`$ docker inspect -f '{{.Name}} - {{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' $(docker ps -aq)` versus `ipconfing`

*Command above gets IP address of a Docker container from host machine or inside a Docker container*

- Normally if you run a container without options it will start and stop immediately, if you want keep it running you can use the command `$ docker run -td container_id`
- Use option -t to allocate a pseudo-TTY session (foreground) -d (daemon mode) automatically detach containers and runs container in background (print container ID) until it is **stopped or killed**

**Piazza Forum Q,**
**VM Seq. – SQL DB resource**

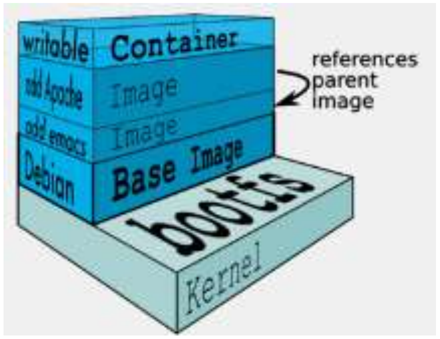
# Docker Images & Deployments Made Easy

**IMAGES** are just templates for docker containers. An ***IMAGE*** is a read-only **TEMPLATE** with instructions (collection of dockerfiles or scripts) for creating a Docker **CONTAINER**

- **TEMPLATES** are instructions & commands to allow you to deploy an Ubuntu VM using "Dockerfile". You can SSH into the VM and run Docker containers. **BASE IMAGES** are what you build your own custom images on top of, Images are **LAYERED**, each layer represents a diff (what changed) from the previous layer
- Build an image based on the ubuntu image, but installs the Apache web server, Python app, as well as the configuration details needed to make app run, Dockerfile (FROM, RUN, COPY, EXPOSE)

## IMAGE LIFECYCLE

docker images shows all images (in /var/lib/docker … "graph" means images)
docker import creates an image from a tarball
docker build creates image from Dockerfile
docker commit creates image from a container, pausing it temporarily if it is running
docker rmi removes an image
docker load loads an image from a tar archive as STDIN, including images and tags
docker save saves an image to a tar archive stream STDOUT with all parent layers, tags & versions



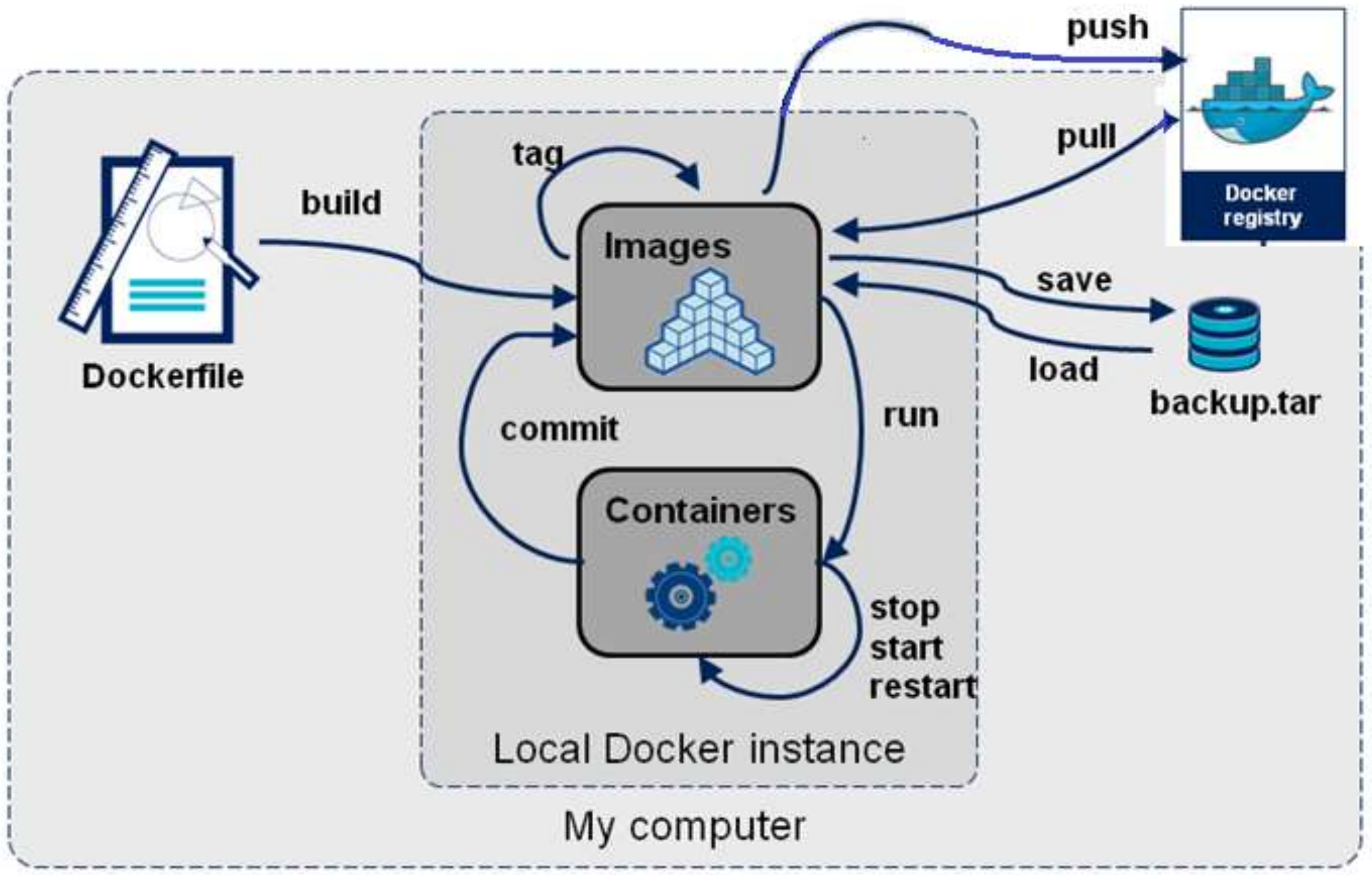
Images can be stored

Search GitHub, Pre-canned Templates for Docker Containers

▪On your Docker host, you can create your own images or use images created by others and published in a Registry
You can use Docker client to manage images, testing …
▪build Docker containers for all the required services and have each test start a new, independent set of containers for just that test

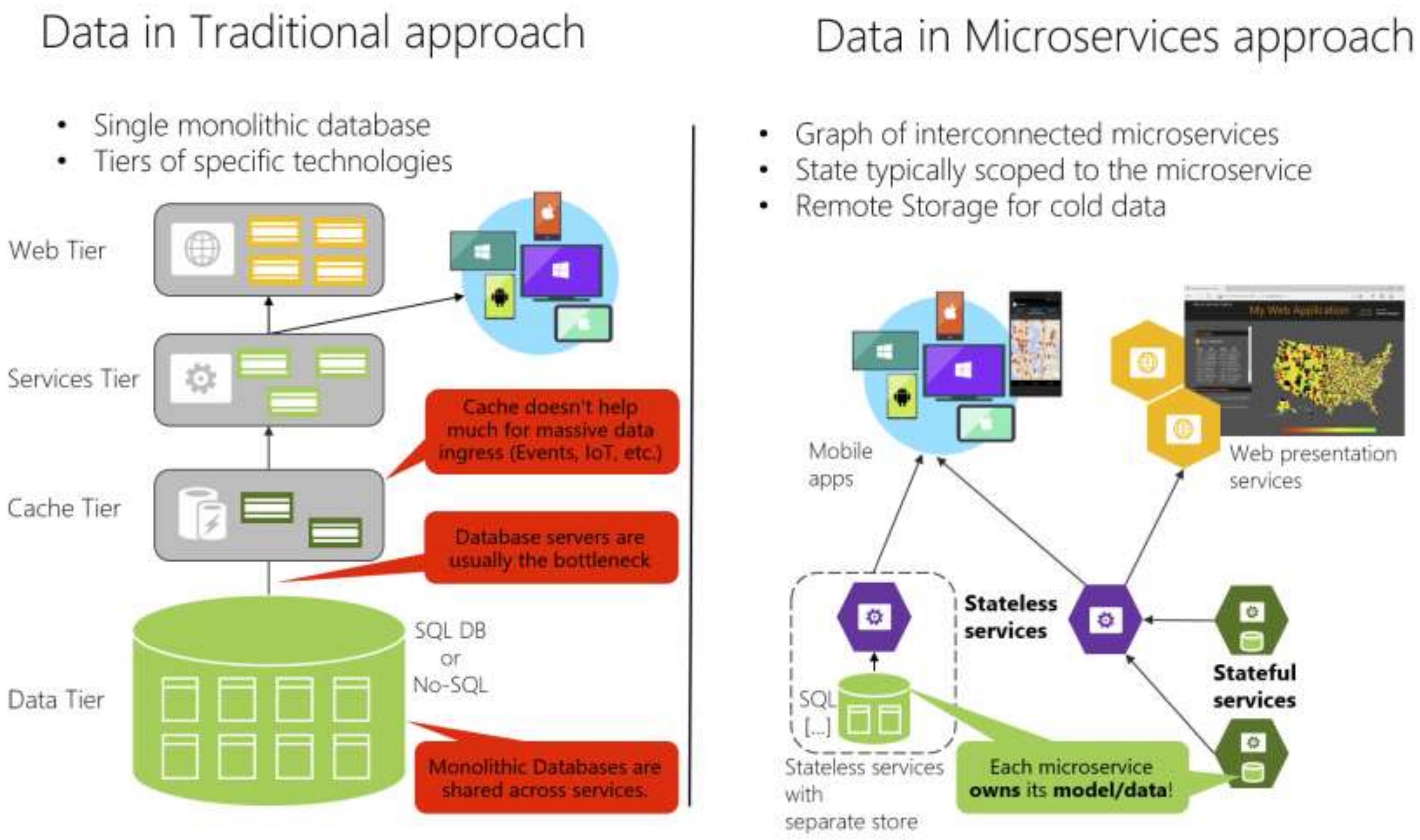
# Docker Lifecycle and Processes

- Docker Repository is public like DockerHub or private, in the Cloud or in your Organization



# Data in Microservices and Bounded Context

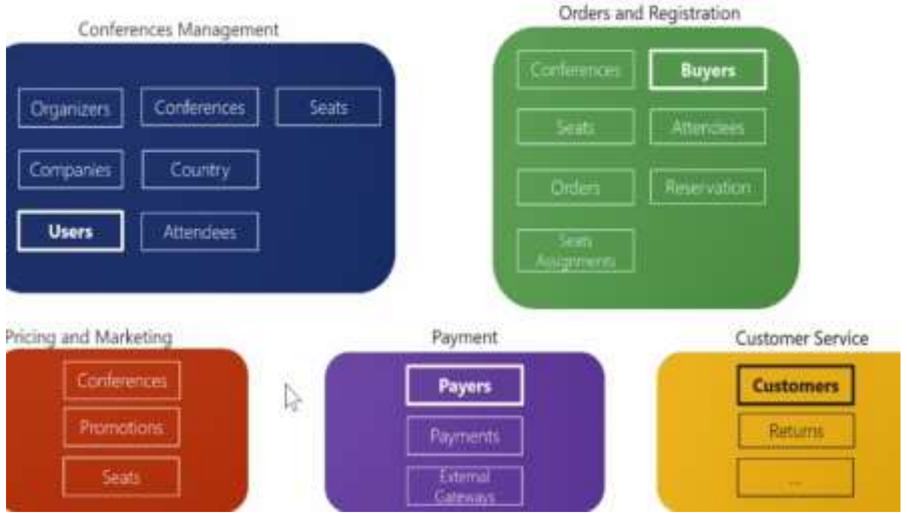
Consider enterprise apps, such as EMPI, CRM, ERP transactional purchase systems, or SalesForce support services … each employs a different Bounded Context (BC) or autonomous subsystem or services that must own its domain model (each call on unique customer entity attributes & data plus logic & behavior). Microservices often use different *kinds* of databases (distributed DBMS challenges)

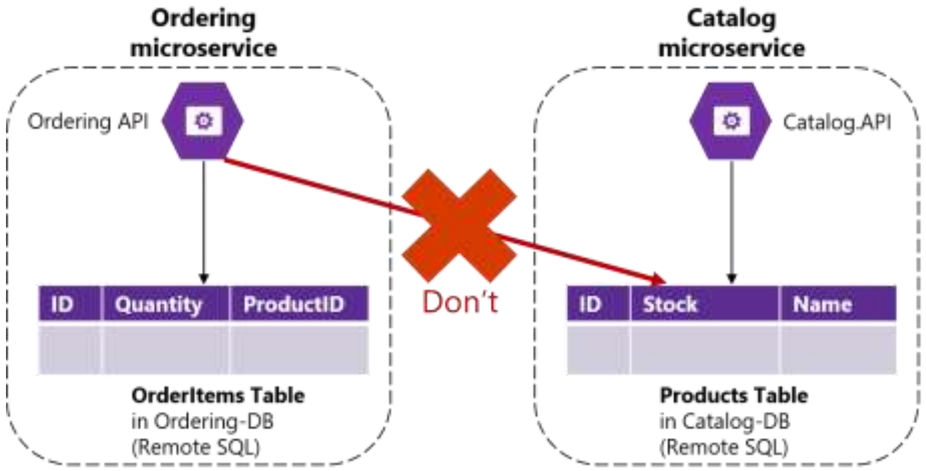

Data in Traditional approach
Single monolithic database
Tiers of specific technologies
Web Tier
Services Tier
Cache Tier
Data Tier
Cache doesn't help much for massive data ingress (Events, IoT, etc.)
Database servers are usually the bottleneck
SQL DB or No-SQL
Monolithic Databases are shared across services.
Data in Microservices approach
Graph of interconnected microservices
State typically scoped to the microservice
Remote Storage for cold data
Mobile apps
Web presentation services
Stateless services
SQL
[...]
Stateless services with separate store
Stateful services
Each microservice owns its model/data!

# Data & Microservice Current Challenges

- How to define the boundaries of each microservice?
  Context driven, For instance, a user can be referred as a user in the identity or membership context, as a customer in a CRM context, as a buyer in an ordering context, and so forth
- How to create queries that retrieve data from several microservices?
  Domain models , A microservice cannot directly access a table in another microservice
- How to achieve consistency across multiple microservices?
  The **database owned by each microservice is private** to that microservice and can only be accessed using its microservice API. Therefore, a challenge presented is how to implement end-to-end business processes while keeping consistency across multiple microservices.

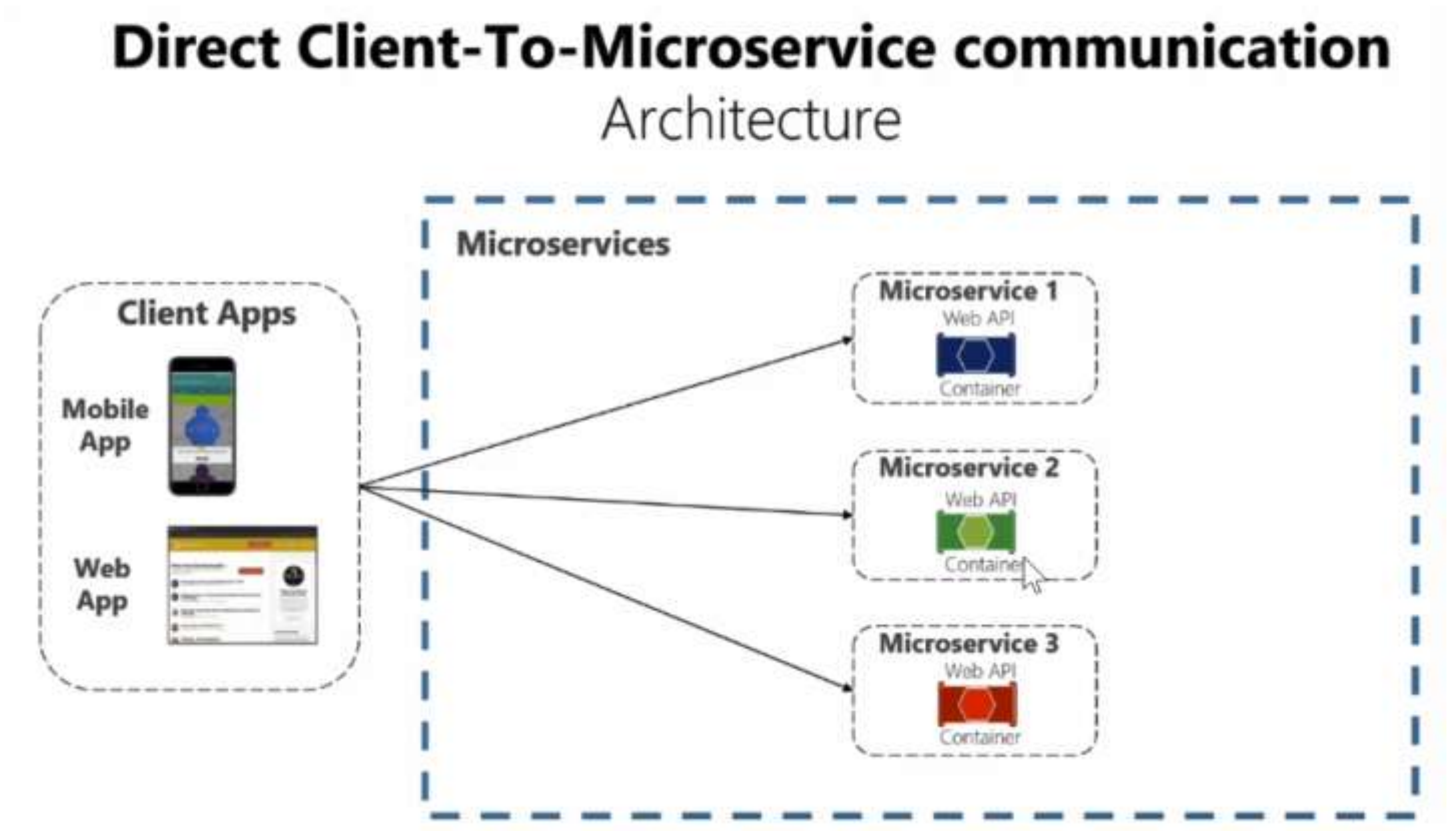
**Modular microservices, each have their own model, defined entities, and depends on requirements for each identified domain in your application (bounded context & API gateway)**





# Data & Microservices Future

https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern



# Docker, Templates, & VM DEMO

**See Docker Cheatsheet with useful commands on week 5 course website**

**SYNTAX MATTERS!**

**“ vs ‘ vs \vs //vs `**
**Typos in Parm’s**
**missing dot . ~/.**

# Demo Requires Installation …

Ensure CygWin, Bash utilities installed, Git Repository, IDE Visual Studio, sign up portal.azure.com, sign up account on hub.docker.com accounts, Install Azure CLI (Windows 10 & MAC)

In order to **create a** Virtual Machine (VM) **that runs a Linux OS** we will do the following:

**Download / install / Create Ubuntu VM or CentOS VM**
- If you know what you are doing, work with any Linux version (Red Hat, RHEL, SUSE, Fedora)
- Use **VMWare Workstation** 11 (12) (Fusion 7 (8)) to create a VM with selected Linux OS
- Use **Windows 10**, Enable the "Windows Subsystem for Linux" optional feature
- Use **Azure** to Provision a Linux VM (request an azure cloud bash shell >_ top rt. portal)

**Install Docker** (CLI, Potal, or Powershell tools)
Run Docker on VM, or Local Host (Win / MAC / Linux OS)
-For example in Windows, MING64 editor $winpty.exe docker run -it ubuntu bash
-For example in CLI, ssh buntu@<azure VM Ubuntu IP address>
Show docker –version, docker images, docker inspect <image>, docker stats
Local Development , Show azure resource group, app (web)

## Reference Links:

https://canvas.instructure.com/courses/1227361/pages/week-4
https://docs.microsoft.com/en-us/cli/azure/sql?view=azure-cli-latest
https://docs.microsoft.com/en-us/azure/sql-database/sql-database-get-started-cli
https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest
https://msdn.microsoft.com/commandline/wsl/install-win10
https://github.com/Azure/azure-quickstart-templates

See week 4 handouts for Ubuntu, Centos, Docker installation

## Install Docker on Your Linux VM

```
$ sudo yum install docker
```

- On Debian/Ubuntu and derivatives.

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

- You can use the `curl` command for installation on several platforms.

```
$ curl -s https://get.docker.io/centos/ | sudo sh
```

- This currently works on:
- Ubuntu; Debian; Fedora; Gentoo.
- Installation on older Mac OsX and Windows requires installation of a special VM.
- For Mac instructions go to http://docs.docker.com/mac/started/
- To start docker engine type

```
$ sudo service docker start
```

Please See Lecture 4 Course Materials!
https://s3.amazonaws.com/deepazure/mckesson-lectures/lecture04/Lecture04CLI_RM.pdf

See week 4 handouts for Ubuntu, Centos, Docker installation

## Docker & Firewalls (Ubuntu)

- If you use the UFW, or Uncomplicated Firewall, on Ubuntu, then you'll need to make a small change to get it to work with Docker.
- Docker uses a network bridge to manage the networking on your containers. By default, UFW drops all forwarded packets. You'll need to enable forwarding in UFW for Docker to function correctly. We can do this by editing the `/etc/default/ufw` file. Inside this file, change:
- **Old UFW forwarding policy** `DEFAULT_FORWARD_POLICY="DROP"`

  To:
- **New UFW forwarding policy** `DEFAULT_FORWARD_POLICY="ACCEPT"`

  Save the update, enable and reload UFW.

```
$ sudo ufw enable
$ sudo ufw reload
```

- If you are testing and developing you may as well leave UFW disabled.

# Search for Images

- Searches your registry for images:

```
$ docker search training
NAME                                   DESCRIPTION                                      STARS               OFFICIAL
training/webapp                                                                         60                      [OK]
training/sinatra                                                                        16
training/postgres                                                                       8                       [OK]
training/namer                                                                          2                       [OK]
training/notes                                                                          2                       [OK]
intelcorp/dl-training-tool             Intel® Deep Learning SDK Training Tool           2
xebialabs/training                     XebiaLabs training material running in Doc...    1                       [OK]
termit/training-apache                 Apache image for the Kubernetes training         1
bgruening/galaxy-training-exome-seq    Galaxy Docker repository for Exome sequenc...    1                       [OK]
kidk/training-images-calc_words        Docker image with calc_words service             0                       [OK]
kidk/training-images-processor         Docker image with processor service              0                       [OK]
training/whoami                        HTTP docker service printing it's containe...    0
kidk/training-images-receiver          Docker image with receiver service               0                       [OK]
krallin/webhooks-training-app                                                           0                       [OK]
hboyce/hboyce-docker-training                                                           0
kidk/training-images-nginx             Docker image with nginx service                  0                       [OK]
kidk/training-images-rabbitmq          Docker image with rabbitmq service               0                       [OK]
```

- Images belong to a namespace. There are several namespaces:

| | |
|---|---|
| Root-like: | ubuntu |
| User: | training/docker-fundamentals-image |
| Self-Hosted: | registry.example.com:5000/my-private-image |

## Port Mappings

- **Containers have their own private IP address**
- You can manually map ports

```
$ docker run -d -p 8080:80 training/webapp python -m SimpleHTTPServer 80
```

`-p` flag takes the form: `host_port:container_port`

This maps port 8080 on the host to port 80 inside the container.

Note that this style prevents you from spinning up multiple instances of the same image (the ports will conflict).

**default network used for Docker containers is 172.17.0.0/16**. If it is already in use on your system, Docker will pick another one. You can ping your container:

```
$ ping 172.17.0.3
64 bytes from 172.17.0.3: icmp_req=2 ttl=64 time=0.085 ms
```

**$ docker inspect** command will find the IP address of the container.

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}'
   <yourContainerID>
172.17.0.3
```

# `docker port` Command

Use the docker port command to find our mapped port

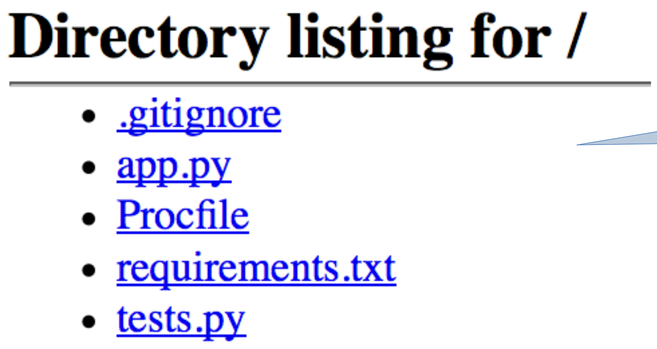
```
$ docker port a721a31e5a2f
5000/tcp -> 0.0.0.0:32772
```

Specify the container ID and the port number for which we wish to return a mapped port

Find the mapped network port using the docker inspect command.

```
$ docker inspect -f "{{ .HostConfig.PortBindings }}"
<yourContainerID>
{"80/tcp":[{"HostIp":"0.0.0.0","HostPort":"49153"}]}
```

To view our Web server, open localhost at port `49153`



Directory listing for /

- .gitignore
- app.py
- Procfile
- requirements.txt
- tests.py

Open Web Browser Localhost:49153

# Docker Networks

## EXPOSE-ing ports

Two of the core concepts of docker are repeatability and portability
Images should able to run on any host and as many times as needed

With **Dockerfiles** you have the ability to map the private and public ports, however, you should **never** map the public port in a Dockerfile. By mapping to the public port on your host you will only be able to have **one instance** of your dockerized app running

```
#Dockerfile  private and public mapping
EXPOSE 80:8080

# private only
EXPOSE 80
```

```
$ docker network ls
NETWORK ID          NAME
d2e4aa487976        bridge
19cceec4933c        host
f8b0c93b7948        none
```

## Container Links

Docker defaults to “**bridge network**” which is created automatically when you install Docker.

Before the Docker networks feature, you could use the Docker link feature to allow containers to discover each other and securely transfer information about one container to another container. With the introduction of the Docker networks feature, you can still create links but they behave differently between default bridge network and user defined networks.

# Docker Tips

Be careful; when you begin with Docker and Windows Containers, you will probably make the following mistakes:

Don't run a daemon in your Dockerfile (docker run –it postgresimage bash …RUN pg_ctl)
You will try to get the IP address from the image instead of the container!

Ensure you get the **container ID** from **docker ps**. if you use the image name, you are asking Docker to show you the IP Address of the image. This, of course, does not make sense, because images do not have IP Addresses.

```
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' <<ContainerID><<NAMES>>
docker run -it --name CoreServerCMD microsoft/windowsservercore cmd.exe
docker attach CoreServerCMD
docker inspect CoreServerCMD
docker inspect -f '{{.Config.Hostname}}'  CoreServerCMD
docker inspect -f '{{.Config.Image}}'  CoreServerCMD
docker inspect -f '{{.NetworkSettings.IPAddress}}'  CoreServerCMD
```

## Deploying a new Windows Container

Use docker **run** command to deploy a new container named **CoreServerCMD** that uses the Windows Server Core image. The **-it** switch denotes an interactive session, and **cmd.exe** means that we want to enter the container inside a new cmd.exe console. **Be careful, docker repository name must be lowercase.**

Basically, the **docker run** translates the **cmd.exe** command within the new Server Core-based container. Now, we have a container named **CoreServerCMD** which is running. We can check with the **docker ps -a** command

# Container Network

- Let's start by creating our new container.

```
$ docker run -d -p 80 training/webapp python -m SimpleHTTPServer 80
72bbff4d768c52d6ce56fae5d45681c62d38bc46300fc6cc28a7642385b99eb5
```

- We've used the -d flag to daemonize the container.
- `-p` flag exposes port 80 in the container.
- We've used the `training/webapp` image, which happens to have Python.
- We've used
- Let's look at our running container.

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
72bbff4d768c ... ... ... ... 0.0.0.0:49153->80/tcp ...
```

- `-p` flag maps a random high port, here 49153 to port 80 inside the container. We can see it in the PORTS column. Python to create a web server at port 80

## Data Containers

- A *data container* is created for the sole purpose of referencing one (or many) volumes.
- It is typically created with a no-op command:

```
$ docker run --name wwwdata -v /var/lib/www busybox true
$ docker run --name wwwlogs -v /var/log/www busybox true
```

- We created two data containers.
- They are using the `busybox` image, a tiny image.
- We used the command `true`, possibly the simplest command in the world!
- We named each container to reference them easily later.

## Using Data Containers

- Data containers are used by other containers thanks to `--volumes-from`.
- Consider the following (fictious) example, using the previously created volumes:

```
$ docker run -d --volumes-from wwwdata --volumes-from wwwlogs
webserver
$ docker run -d --volumes-from wwwdata ftpserver
$ docker run -d --volumes-from wwwlogs pipestash
```

- The first container runs a webserver, serving content from `/var/lib/www` and logging to `/var/log/www.`
- The second container runs a FTP server, allowing to upload content to the same `/var/lib/www path`.
- The third container collects the logs, and sends them to `logstash`, a log storage and analysis system.

## Sharing a single host file with a container

- The same -v flag can be used to share a single file.

```
$ echo 4815162342 > /tmp/numbers
$ docker run -t -i -v /tmp/numbers:/numbers ubuntu bash
root@274514a6e2eb:/# cat /numbers
4815162342
```

- All modifications done to `/numbers` in the container will also change `/tmp/` numbers on the host!
- It can also be used to share a *socket*.

```
$ docker run -t -i -v /var/run/docker.sock:/docker.sock ubuntu
bash
```

- This pattern is frequently used to give access to the Docker socket to a given container.

## Sharing directories between host and a container

```
$ docker run -t -i -v /src/webapp:/var/www/html/webapp
ubuntu /bin/bash
```

- This will mount host directory `/src/webapp` into the container directory `/var/www/html/webapp`.
- It defaults to mounting read-write but we can also mount read-only.

```
$ docker run -t -i -v /src/webapp:/var/www/html/webapp:ro ubuntu
/bin/bash
```

- Those volumes can also be shared with `--volumes-from`.

# Docker, Templates, & VM DEMO

SYNTAX MATTERS!

“ vs ‘ vs \vs //vs `
Typos in Parm’s
missing dot .   ~/.

# Build your own Images, **Dockerfile**

- `Dockerfile` is a file which holds Docker image definitions.
- `Dockerfile` is the "build recipe" for a Docker image.
- File contains instructions telling Docker how an image is constructed.
- The `docker build` command builds an image from a `Dockerfile`

```
# Download from https://github.com/docker-training/staticweb
        FROM ubuntu:14.04
        MAINTAINER Docker Education Team <education@docker.com>
        RUN apt-get update
        RUN apt-get install -y nginx
        RUN echo 'Hi, I am in your container' \
        >/usr/share/nginx/html/index.html
        CMD [ "nginx", "-g", "daemon off;" ]
        EXPOSE 80
```

FROM specifies a source image for our new image. It's mandatory.

MAINTAINER tells us who maintains this image.

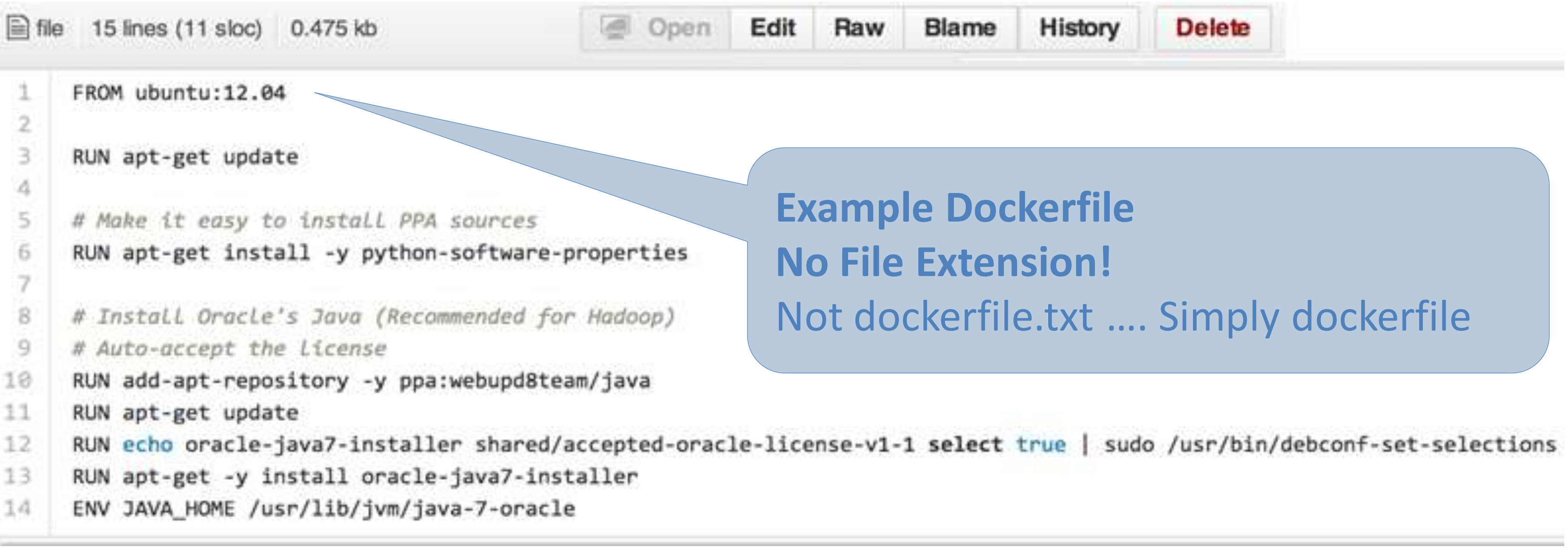
Each RUN instruction executes a command to build our image.

CMD defines the default command to run when container is launched from this image.

EXPOSE lists the network ports to open when a container is launched from this image.

# Dockerfile

- Like a Makefile (shell script with keywords)
- Extends from a Base Image
- Results in a new Docker Image
- Imperative, not Declarative
- Docker file lists the steps needed to build an images
- **docker build** is used to run a Docker file
- Can define default command for docker run, ports to expose, etc



```
FROM ubuntu:12.04

RUN apt-get update

# Make it easy to install PPA sources
RUN apt-get install -y python-software-properties

# Install Oracle's Java (Recommended for Hadoop)
# Auto-accept the license
RUN add-apt-repository -y ppa:webupd8team/java
RUN apt-get update
RUN echo oracle-java7-installer shared/accepted-oracle-license-v1-1 select true | sudo /usr/bin/debconf-set-selections
RUN apt-get -y install oracle-java7-installer
ENV JAVA_HOME /usr/lib/jvm/java-7-oracle
```

**Example Dockerfile**
**No File Extension!**
Not dockerfile.txt …. Simply dockerfile

## `Dockerfile` Usage

- `Dockerfile` instructions are executed in order
- Each instruction creates a new layer in the image
- Instructions are cached. If no changes are detected then the instruction is skipped and the cached layer used
- **FROM instruction MUST be the first non-comment instruction**
- You can only have one FROM and ENTRYPOINT command
- Lines starting with # are treated as comments

- We use the docker build command to build images

```
    $ docker build -t web .
```

- The -t flag tags an image
- The . indicates the location of the `Dockerfile` being built
- We can also build from other sources, such as a GitHub  repository

```
$ docker build -t web https://github.com/docker-training/staticweb.git
```

# Dockerfile

## Dockerfile Instructions

FROM instruction MUST be the first non-comment instruction

.dockerignore

FROM Sets the Base Image for subsequent instructions.
MAINTAINER (deprecated - use LABEL instead) Set the Author field of the generated images.
RUN execute any commands in a new layer on top of the current image and commit the results.
CMD provide defaults for an executing container.
EXPOSE informs Docker that the container listens on the specified network ports at runtime. NOTE: does not actually make ports accessible.
ENV sets environment variable.
ADD copies new files, directories or remote file to container. Invalidates caches. Avoid ADD, use COPY instead
COPY copies new files or directories to container. Note that this only copies as root, so you have to chown manually regardless of your USER / WORKDIR setting. See https://github.com/moby/moby/issues/30110
ENTRYPOINT configures a container that will run as an executable.
VOLUME creates a mount point for externally mounted volumes or other containers.
USER sets the user name for following RUN / CMD / ENTRYPOINT commands.
WORKDIR sets the working directory.
ARG defines a build-time variable.
ONBUILD adds a trigger instruction when the image is used as the base for another build.
STOPSIGNAL sets the system call signal that will be sent to the container to exit.
LABEL apply key/value metadata to your images, containers, or daemons.

## FROM and MAINTAINER instructions

- Specifies the source image to build this image.
- Must be the first instruction in the `Dockerfile`.
- (Except for comments: it's OK to have comments before FROM.)
- Can specify a base image:

```
FROM ubuntu
```

- An image tagged with a specific version:

```
FROM ubuntu:12.04
```

- A user image:

```
FROM training/sinatra
```

- Or self-hosted image:
- `FROM localhost:5000/funtoo`
- The MAINTAINER instruction tells you who wrote the Dockerfile.

```
MAINTAINER Docker Education Team <education@docker.com>
```

## RUN Instruction

- The RUN instruction can be specified in two ways.
- With shell wrapping, which runs the specified command inside a shell, with `/bin/sh -c`:

```
RUN apt-get update
```

- Or using the exec method, which avoids shell string expansion, and allows execution in images that don't have `/bin/sh`:

```
RUN [ "apt-get", "update" ]
```

- RUN will do the following:
  - Execute a command, Record changes made to the filesystem, Install libraries, packages, and various files.
- RUN will NOT do the following:
  - Record state of *processes*, Automatically start daemons.
- To start something automatically when the container runs, you should use CMD and/or ENTRYPOINT.

## EXPOSE Instructions

- The EXPOSE tells Docker what ports are to be published in this image.

```
EXPOSE 8080
```

- All ports are private by default.
- The Dockerfile does not control if a port is publicly available.
- When you docker run -p <port> ..., that port becomes public. (Even if it was not declared with EXPOSE.)
- When you docker run -P ... (without port number), all ports declared with EXPOSE become public.
- A *public port* is reachable from other containers and from outside the host. A *private port* is not reachable from outside.

## ADD Instruction

- The ADD instruction adds files and content from your host into the image.

```
ADD /src/webapp /opt/webapp
```

- This will add the contents of the `/src/webapp/` directory to the `/opt/webapp` directory in the image.
- Note: `/src/webapp/` is not relative to the host filesystem, but to the directory containing the Dockerfile. Otherwise, a Dockerfile could succeed on host A, but fail on host B.
- The ADD instruction can also be used to get remote files.

```
ADD http://www.example.com/webapp /opt/
```

- This would download the webapp file and place it in the /opt directory.
- ADD is cached. If you recreate the image and no files have changed then a cache is used.
- If the local source is a zip file or a tarball it'll be unpacked to the destination. Sources that are URLs and zipped will not be unpacked.
- Any files created by the ADD are owned by root with permissions of 0755.

## VOLUME instruction

- The VOLUME instruction will create a data volume mount point at the specified path.

```
VOLUME [ "/opt/webapp/data" ]
```

- Data volumes bypass the union file system.
- In other words, they are not captured by `docker` commit.
- Data volumes can be shared and reused between containers.
- It is possible to share a volume with a stopped container.
- Data volumes persist until all containers referencing them are destroyed.

## Example of `Dockerfile` usage, Tomcat

- With what we learned about `Dockerfile`, let us create a custom container.
- We will login into Docker hub and fetch an existing image with Tomcat

```
# sudo docker login
# sudo docker pull tomcat:jre8
```

- Next, let us create a new directory, call it `ztomcat`

```
# mkdir ztomcat
# cd ztomcat
```

- In that directory let us create a simple HTML file `index.html`:

```
<!DOCTYPE HTML><html lang="en">
<head>
<meta charset="UTF-8">
<title>cscie90 Class</title>
</head>
<body>
<p
<button onclick="this.innerHTML=Date()">The time is?</button>
</body>
</html>
```

## My Tomcat Container

- Next in the directory `ztomcat` create the file `Dockerfile` with the following content:

```
FROM tomcat:jre8
MAINTAINER "Zoran<zdjordj@fas.harvard.edu>"
ADD index.html /usr/local/tomcat/webapps/examples/
```

- Let us see which images exist

```
# sudo docker images
```

- While in directory `ztomcat`, let us now create new image:

```
# sudo docker build -t zdjordje/ztomcat .
# sudo docker images
# sudo docker run -i -t -p 8888:8080 zdjordje/ztomcat
```

- If we open the browser inside our CentOS VM, we should see our page.
- Finally we could push new image to Docker hub

```
# sudo docker push zdjordje/ztomcat
```

- You may also delete local image and bring one from repository

```
# sudo docker rmi –f zdjordje/ztomcat
# sudo docker pull zdjordje/ztomcat
```

# Creating a Container from local image

- We've got an image, some source code and now we can add a container to run that code.

```
$ docker run -d -v $(pwd):/opt/namer -w /opt/namer \
-p 80:9292 training/namer rackup
```

- We are passing some *flags* as arguments to the `docker run` command to control its behavior:
- `-d` flag indicates that the container should run in daemon mode (in the background).
- `-v` flag provides volume mounting inside containers.
- `-w` flag sets the working directory inside the container.
- `-p` flag maps port 9292 inside the container to port 80 on the host.
    - We've launched the application with the `training/namer` image and Ruby `rackup` command.

## Working with Volumes

- Docker volumes can be used to achieve many things, including:
  - Bypassing the copy-on-write system to obtain native disk I/O performance.
  - Bypassing copy-on-write to leave some files out of docker commit.
  - Sharing a directory between multiple containers.
  - Sharing a directory between the host and a container.
  - Sharing a *single file* between the host and a container.
- Volumes are special directories in a container
- Volumes can be declared in two different ways.
- Within a Dockerfile, with a VOLUME instruction.

```
VOLUME /var/lib/postgresql
```

- On the command-line, with the -v flag for docker run.

```
$ docker run -d -v /var/lib/postgresql training/postgresql
```

- In both cases, `/var/lib/postgresql` (inside the container) will be a volume.

## Volumes speak to the host OS

- Volumes act as passthroughs to the host filesystem.
  - The I/O performance on a volume is exactly the same as I/O performance on the Docker host.
  - When you docker commit, the content of volumes is not brought into the resulting image.
  - If a RUN instruction in a Dockerfile changes the content of a volume, those changes are not recorded neither.

## Volumes can be shared across containers

- You can start a container with *exactly the same volumes* as another one.
- The new container will have the same volumes, in the same directories.
- They will contain exactly the same thing, and remain in sync.
- Under the hood, they are actually the same directories on the host anyway.

```
$ docker run --name alpha -t -i -v /var/log ubuntu bash
root@99020f87e695:/# date >/var/log/now
```

- In another terminal, let's start another container with the same volume.

```
$ docker run --volumes-from alpha ubuntu cat /var/log/now
Fri May 30 05:06:27 UTC 2014
```

- Volumes exist independently of containers
- If a container is stopped, its volumes still exist and are available.
- In the last example, it doesn't matter if container `alpha` is running or not.

## Share a host volume with multiple containers

- We could do the following

```
$ docker run --name data -v /tmp/volume:/volume busybox true
$ docker run --volumes-from data busybox touch /volume/hello
$ ls -l /tmp/volume/hello
-rw-r--r-- 1 root root 0 May 29 23:27 /tmp/volume/hello
```

- We created a data container named `data`, for `/volume`, mapped to `/tmp/` volume on the host.
- We created another container, using this volume, and creating a file there.
- From the host, we checked that the file appeared as expected.
- We can check volumes defined by an image using `docker inspect`:

```
$ # docker inspect training/datavol
[{
    "config": {
    . . .
    "Volumes": {
    "/var/webapp": {}
    },
```

# Appendix

## Azure Data Science VM

The Data Science Virtual Machine (DSVM) is a customized VM image on Microsoft's Azure cloud built specifically for doing data science. It has many popular data science and other tools pre-installed and pre-configured to jump-start building intelligent applications for advanced analytics. It is available on Windows Server and on Linux. We offer Windows edition of DSVM on Server 2016 and Server 2012. We offer Linux edition of the DSVM on Ubuntu 16.04 LTS and on OpenLogic 7.2 CentOS-based Linux distributions.

## For those with interest in Healthcare / Clinical Apps … Try these public images!

```
$ docker pull lelagina/clinical_pipeline
$ docker pull lingdu0001/clinic
$ docker pull clinicalgraphics/miniconda3
```

**Reference Links:**

https://hub.docker.com/search/?isAutomated=0&isOfficial=0&page=1&pullCount=0&q=clinic&starCount=0

https://docs.microsoft.com/en-us/azure/machine-learning/data-science-virtual-machine/overview

# Appendix

Install Visual Studio Code tool to work with Azure
Use azure resource snippet … for templates, JSON schemae, resource properties
OR use Visual Studio … which has all those azure tools embeded
OR use GUI @portal.azure.com and native IDE's (CLI, Powershell, Advanced Tools – "Kudu")

Create a SQL server:
```
az sql server create -n xxx -u xxx -p xxx -g yourresourcegroup -l location
```
Create SQL database to run on server:
```
az sql db create -n xxx -s xxx -g xxx
```
Create your firewall-rule to allow remote access:
```
az sql server firewall-rule create xxx
```

Connect to your DB, create simple table, add 2 rows
Can use IDE (SQL Server Mgmt Studio, VS)
Query data & output report:

**Reference Links:**
Deploy Azure resources through the Azure Resource Manager with community contributed templates
https://github.com/Azure/azure-quickstart-templates
https://azure.microsoft.com/en-us/resources/templates/

App (Web) Services Resources:
https://docs.microsoft.com/en-us/azure/sql-database/
https://docs.microsoft.com/en-us/azure/app-service/
https://docs.microsoft.com/en-us/azure/app-service/app-service-web-get-started-java
https://docs.microsoft.com/en-us/azure/app-service/app-service-web-tutorial-dotnet-sqldatabase

# Appendix

## SQL Server Lab

https://github.com/docker/labs/tree/master/windows/sql-server

- You'll need Docker running on Windows. You can install Docker for Windows on Windows 10, or follow the Windows Container Lab Setup to install Docker on Windows locally, on AWS and Azure.
- You should be familiar with the key Docker concepts, and with Docker volumes:
- Docker concepts
- Docker volumes

- Part 1 - Building the Dacpac
- Part 2 - Building the SQL Server Image
- Part 3 - Running the SQL Server Container
- Part 4 - Upgrading the SQL Server Database