Containers on .NET Core Anses – 2/3

Robert Rozas Navarro
Customer Engineer
Azure-App Dev Domain

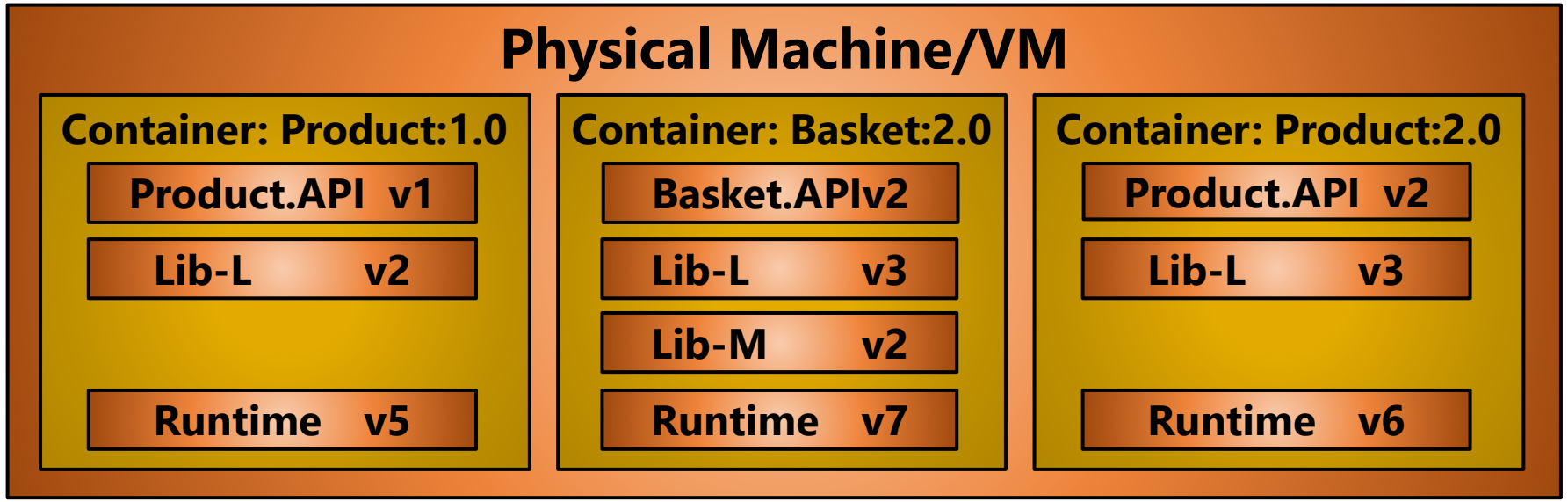
Containers Fundamentals

What is a Container?

Microsoft Services

What is a Container?

- Portable unit of deployment
- Packs application code and dependencies together into single unit
- Virtualization without the need of a full virtual machine
  - Slice up the OS to run multiple apps on a single OS
  - Each container runs in isolated memory, but shares the kernel of underlying host
- Typically run one service per container (container and app share lifecycle)

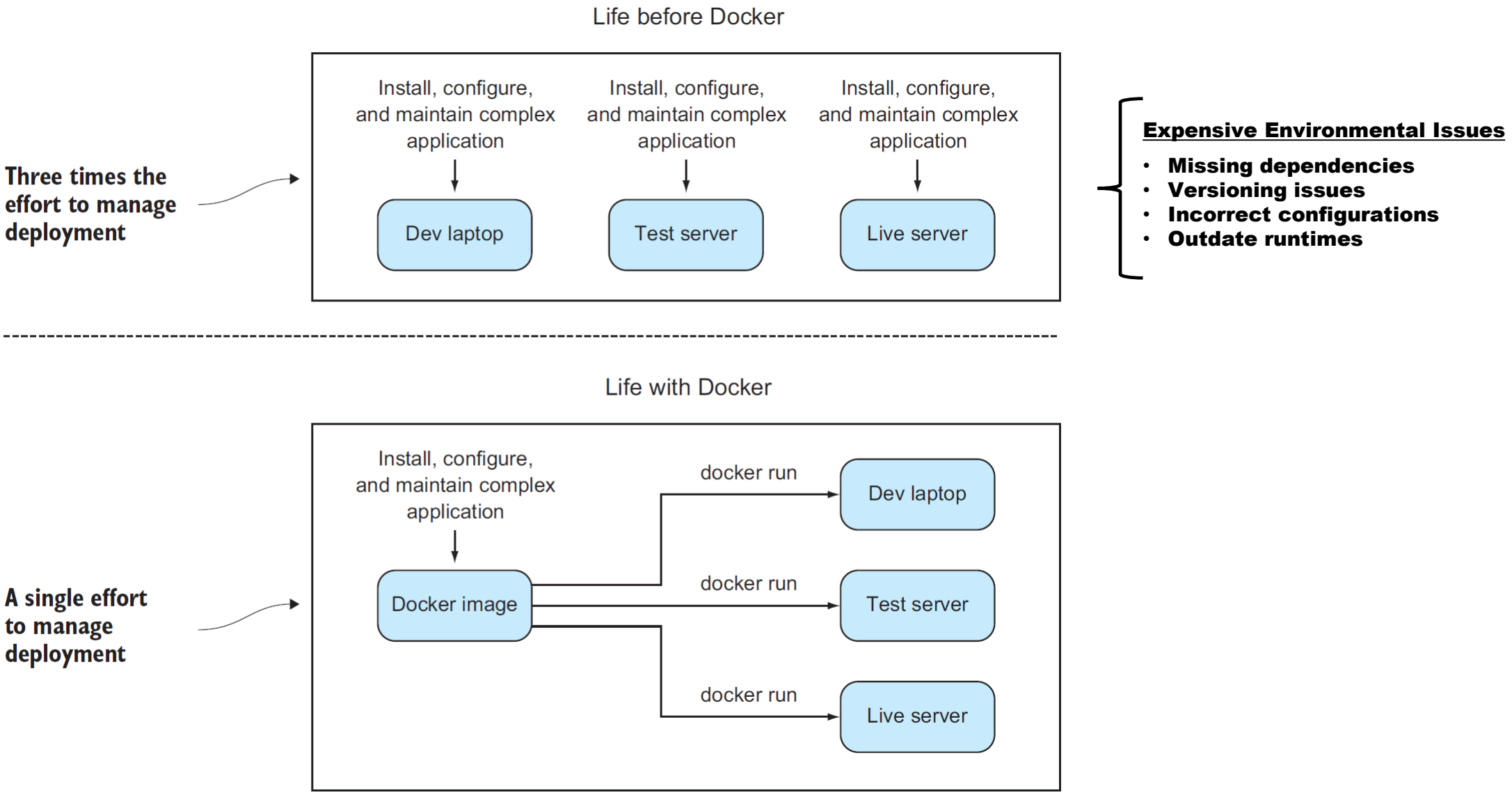


Microsoft Confidential

What is Docker?

- *Docker* is the company driving the software container movement
  - In a short time, it has become the de facto standard for packaging, deploying and running distributed and cloud native platforms
- Docker is a technology stack…
  - An open *platform* that enables you to "build, ship, and run any app, anywhere"
  - A container *format*
  - A set of *tools* for creating and running application in containers
  - Includes open source and (for-purchase) enterprise offerings
- Docker has become a standard for solving one of the costliest aspects of software: deployment

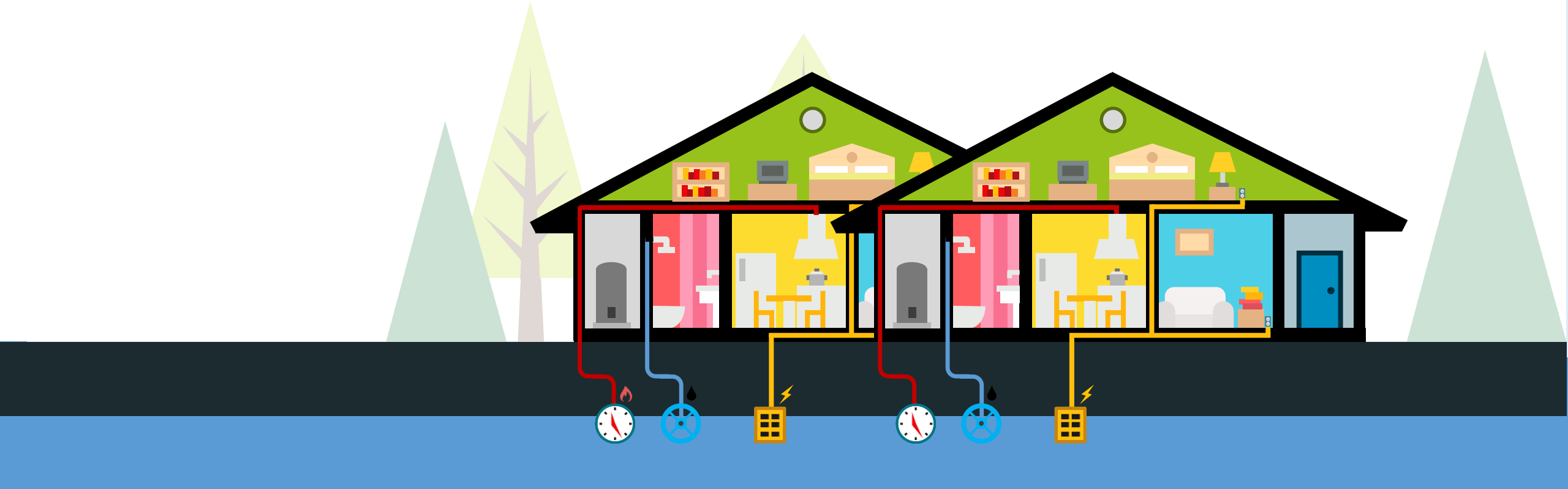
Microsoft Confidential



Microsoft Confidential

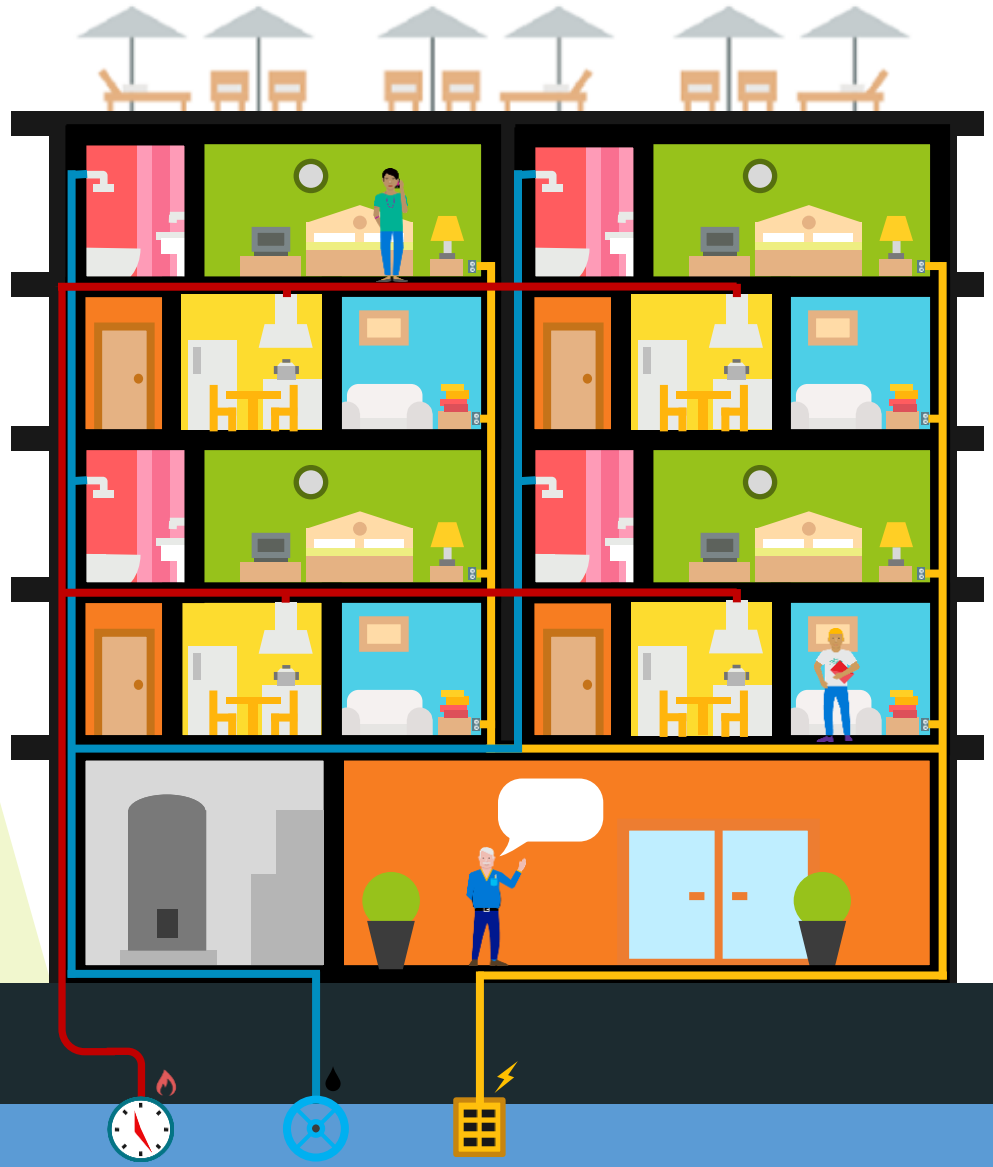
Virtual Machine (VM) vs. Container

- VMs are single, isolated entities residing on the same host
- VMs don’t share resources
- Each supports a full operating system
- Think of single houses on a block



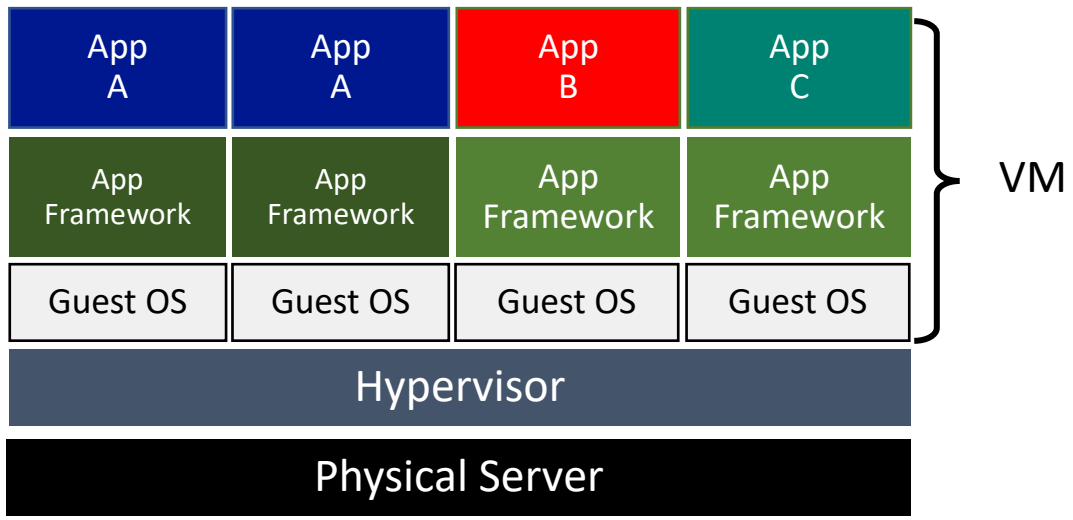
Virtual Machine (VM) vs. Container

Containers are like apartments, they have their individual resources but share core resources



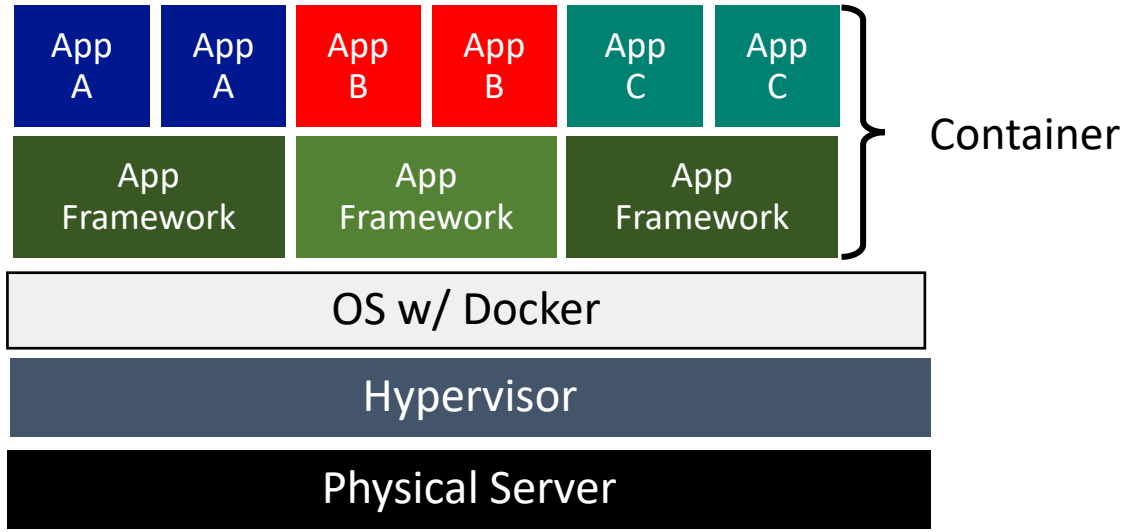
At the Plumbing Level

Virtual machines = Hardware virtualization



Each VM has its own full, independent OS, but share the core hardware

Containers = Operating System virtualization



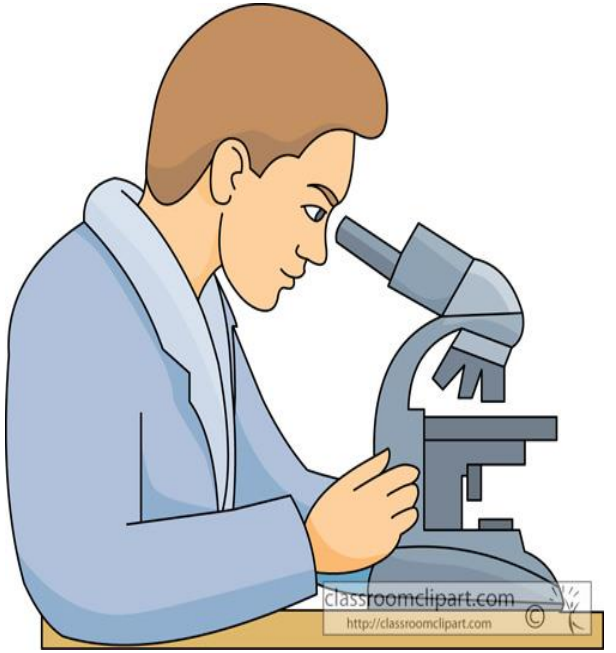
Each container shares hardware, kernel, OS and libraries

- They run on Linux, Windows 2016, 2019, 10 and in Azure

Microsoft Confidential

Inside a Container

- Inside a container, it looks like you are inside a freshly installed physical computer or a virtual machine
- Shares the kernel of the host OS for virtualized access to CPU, memory, network, registry and tasks such as running processes, managing hardware devices and handling interrupts
- Has a layer of protection between host and container and to isolate from other containerized processes



Isolation versus density

	Physical Machine	VM	Container
Hardware	Not shared	Shared	Shared
OS Kernel	Not shared	Not shared	Shared
System Resources (ex: File System)	Not shared	Not shared	Not shared

What Problems Do Containers Solve?

- Guarantees consistency across dev, test and prod environments – everything is self-contained
  - Provides portability
  - Eliminates the “Works on my machine” dilemma
- Increases Productivity
  - Less time setting up environments
  - Eliminates debugging environment-specific issues
- Smaller footprint than VMs
  - Increase server consolidation and density per host
- Isolation
  - Each container has separate slice of OS, CPU and memory without affecting other containers
- Performant and quick start-up

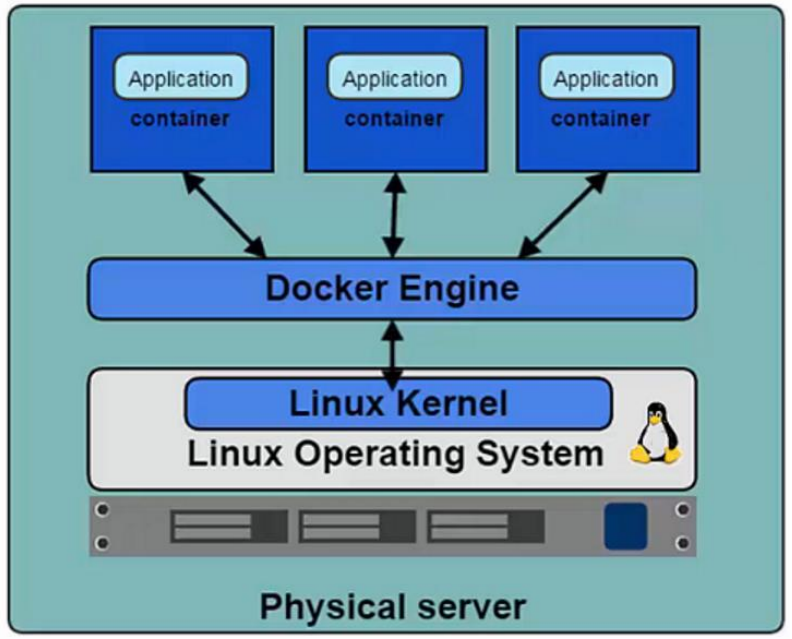
Containers Fundamentals

How do containers work?

Microsoft Services

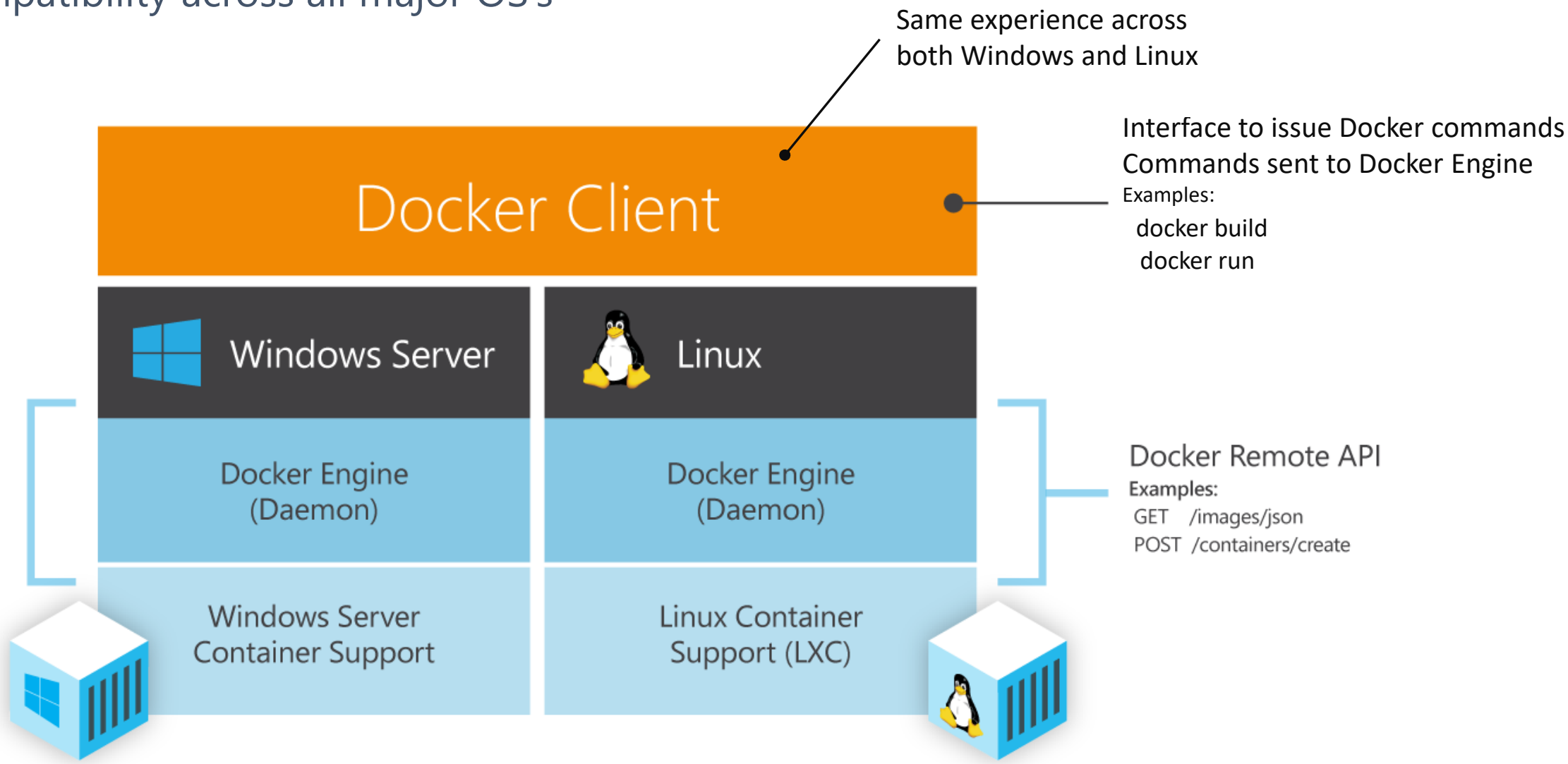
Key Docker Components

- Host
  - VM/Physical machine running the Docker daemon
- Docker Client
  - Application that interacts with Docker
- Docker Daemon (Engine)
  - Software that enables containers to be built and ran
- Docker Image
  - Ordered collection of read-only filesystem layers that produce a container
- Docker Container
  - Running instance of a Docker image
- Docker Registry
  - Online library to store Docker images



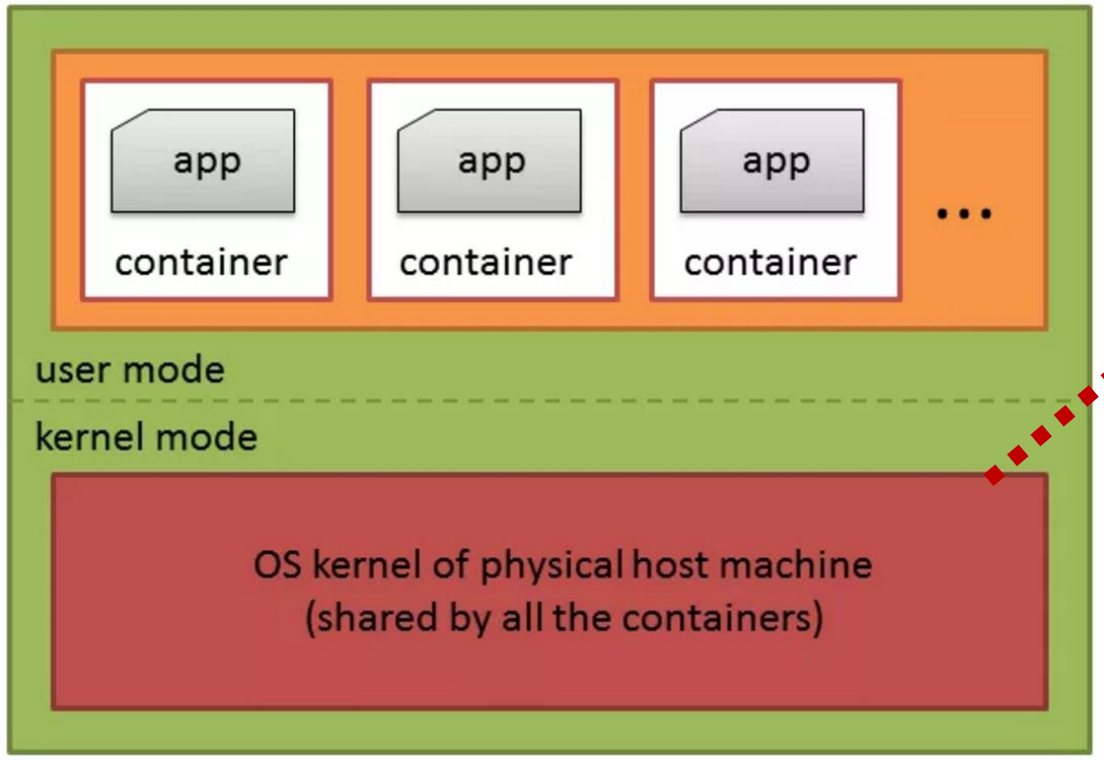
Linux vs Windows Containers

- Compatibility across all major OS's



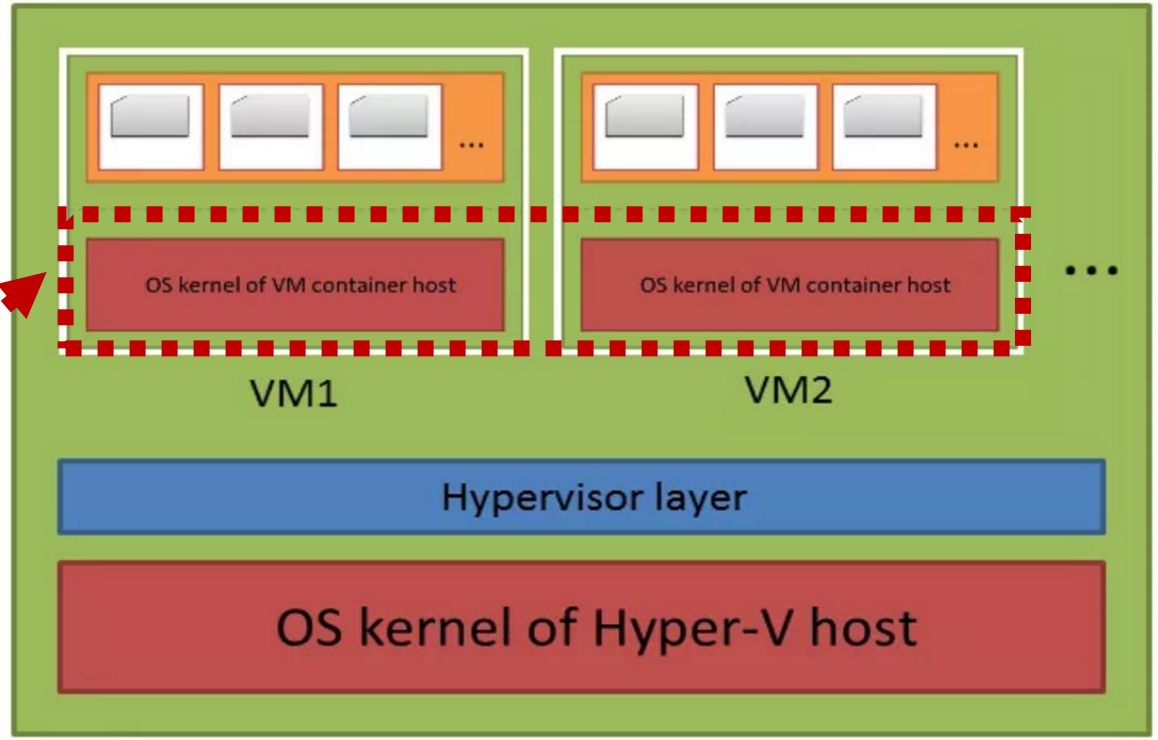
Windows Containers

Window Server Containers



Share O/S Kernel

Window Hyper-V Containers

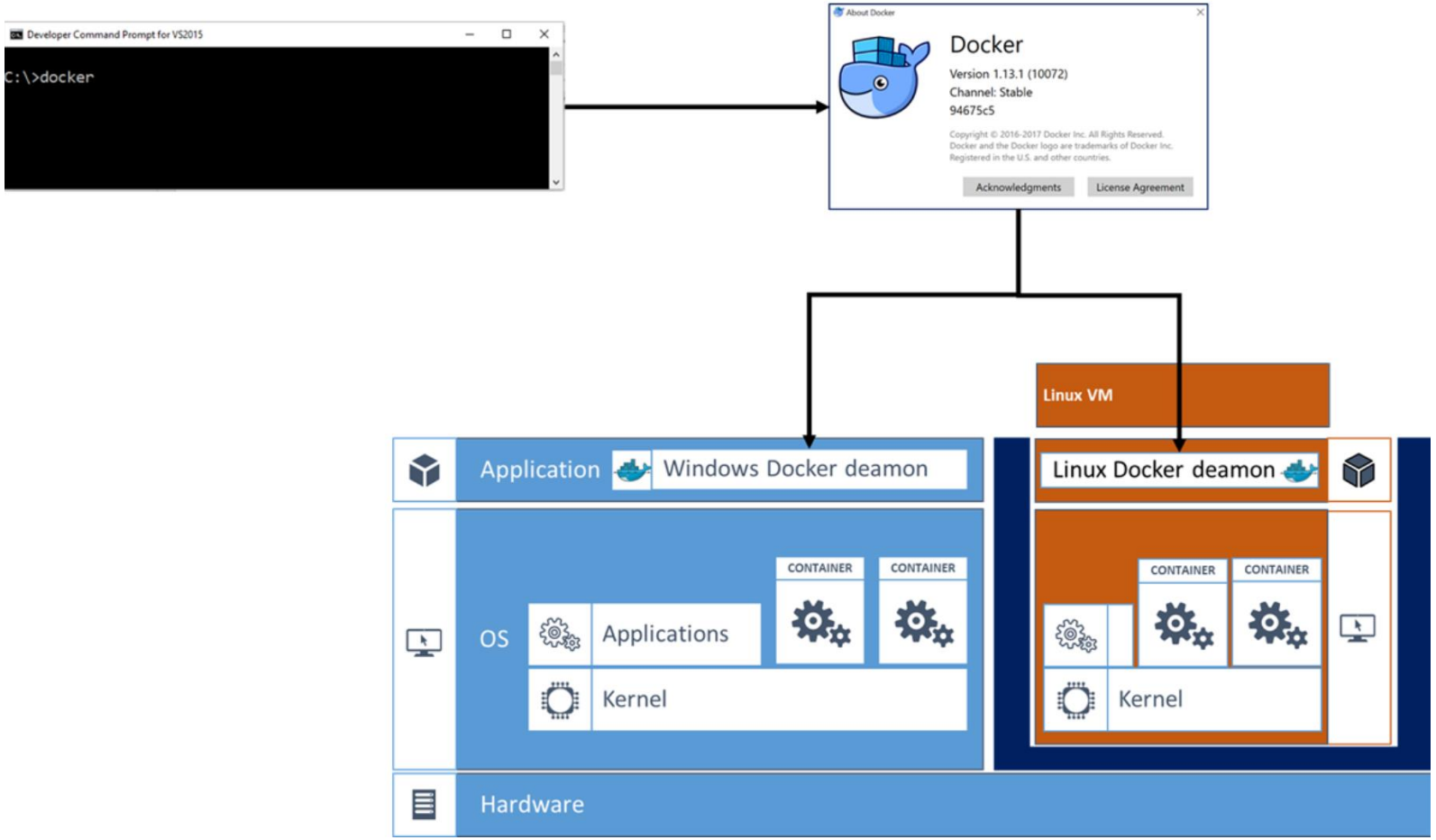


Isolated O/S Kernel

Microsoft Confidential

Docker for Windows Client

- Simple, integrated Docker development environment for Windows 10
- Installs Hyper-V Moby Linux VM – can run Linux containers on Windows



Demonstration

Explore Docker For Windows

Docker Components in Action

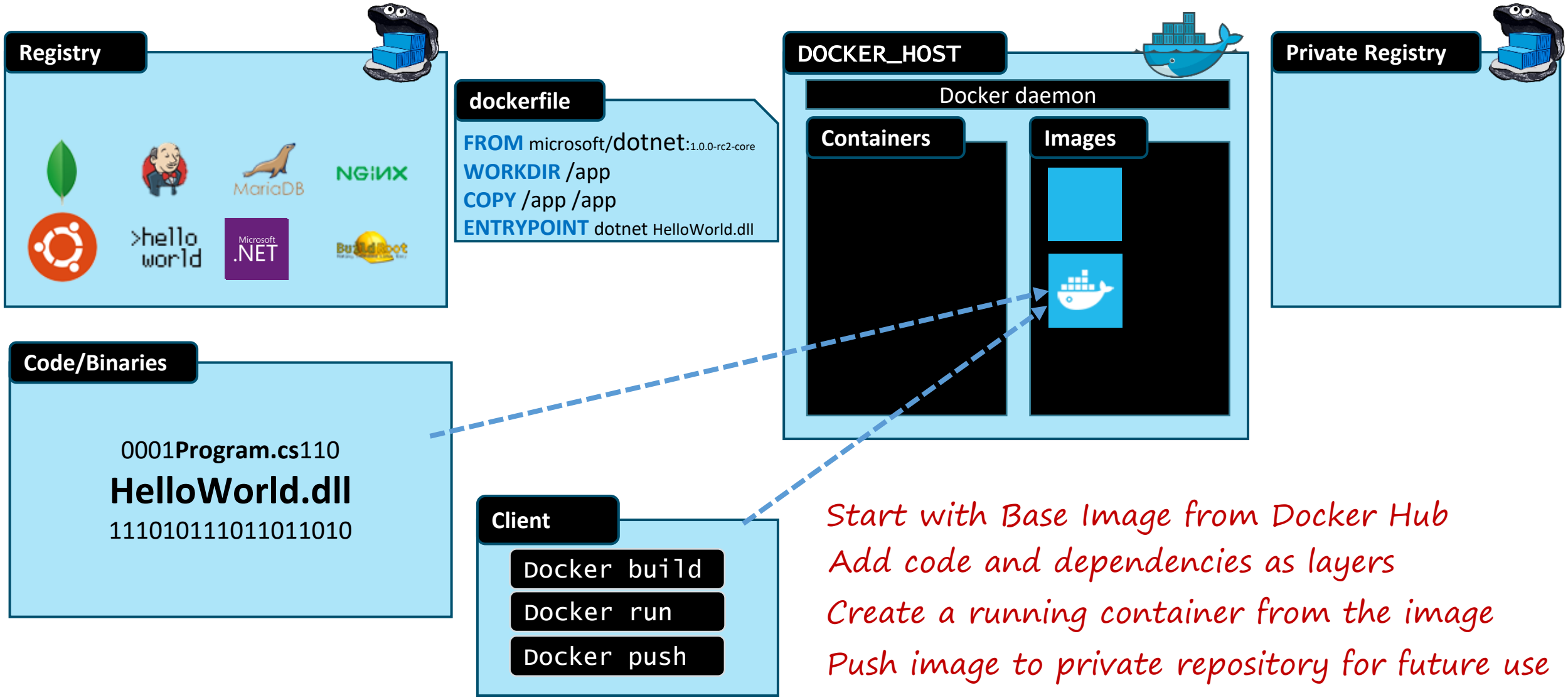


Image Registries

What is a registry?

- Location to store container images
  - **Push** images into a registry
  - **Pull** images from a registry
  - **Search** images within a registry

Docker Hub and Docker Store

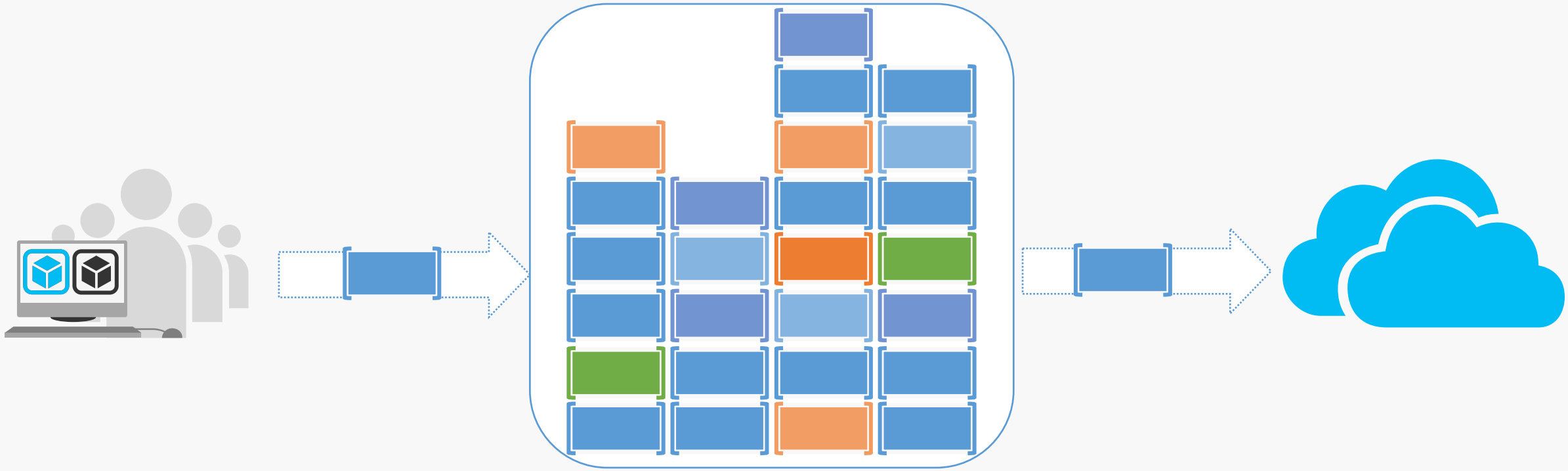
- Public, official and private image repositories

Docker Trusted Registry (DTR)

- Enterprise grade private registries

Azure Container Registry (ACR)

- Store both Windows and Linux images in Azure
- Same API and Tools as Docker Hub/Store/Registry

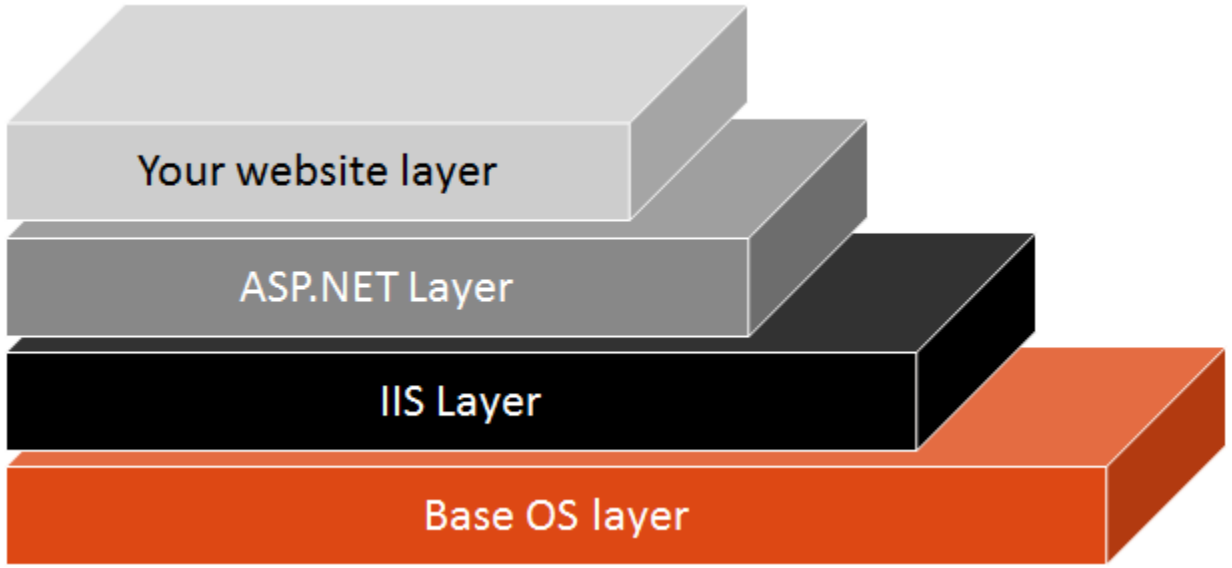


Demonstration

Explore Docker Registry

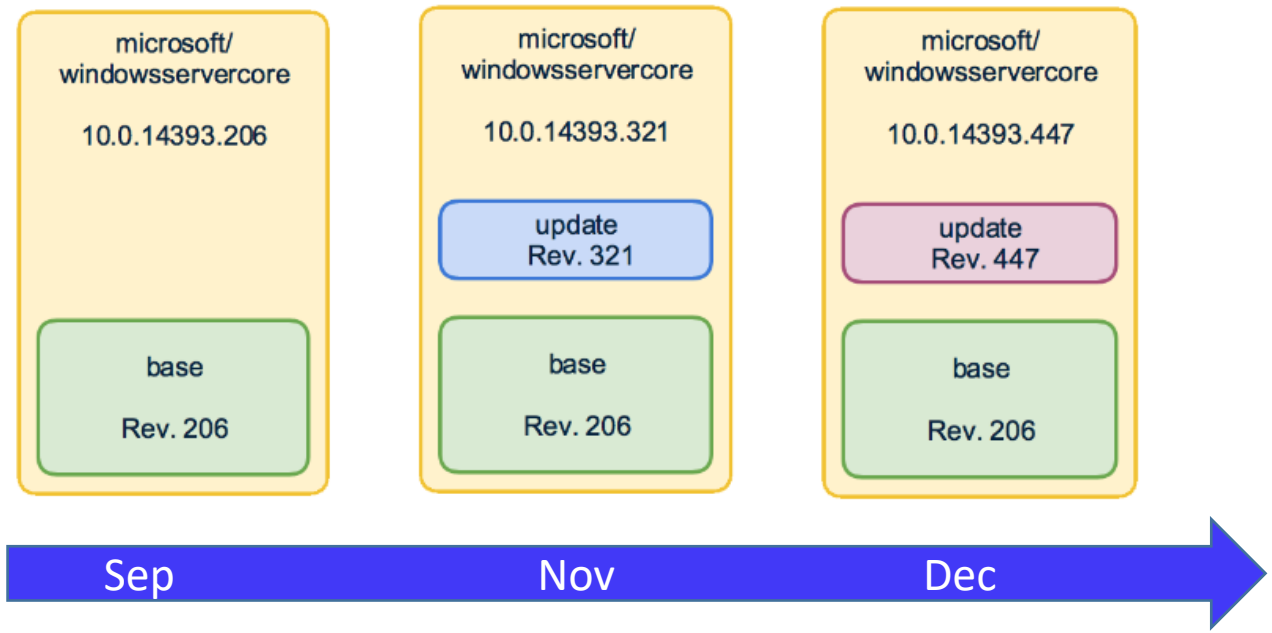
Container Image

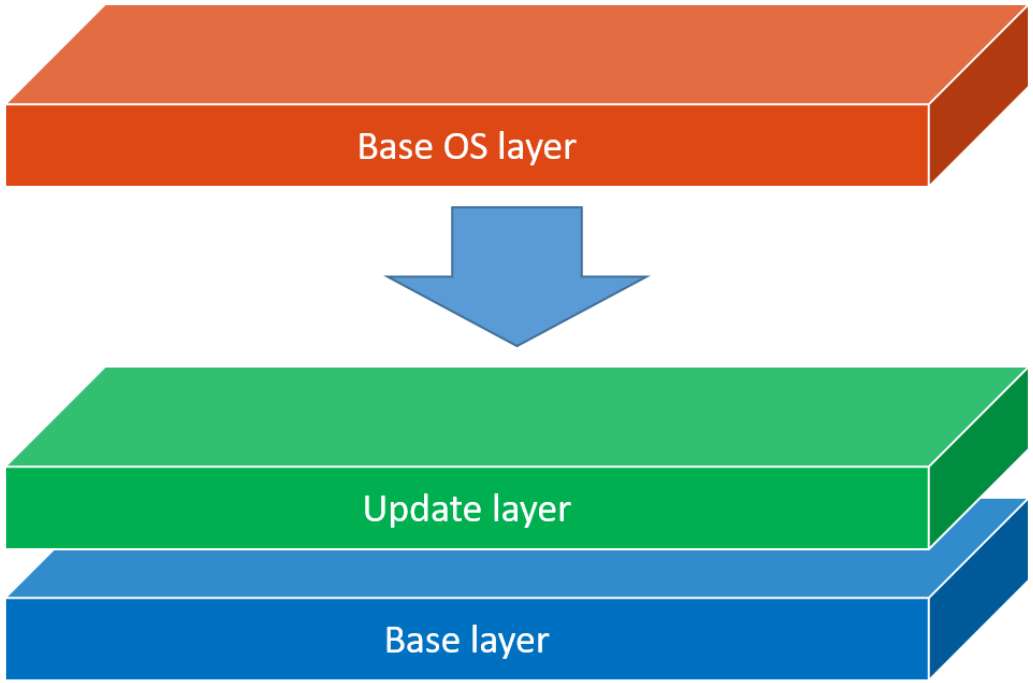
- Multi-layered snapshot of all components that make up a container environment: *OS, IIS, .NET Framework and the app*
- Base OS image pulled from a registry, typically, Docker Hub
- Additional components are layered on top
- Each layer is immutable (read-only), except for the top-most layer
- Each layer represents an instruction in the image's Dockerfile
- Template for a container



Base OS Image

- The bottom-most layer
  - Contains core elements of operating system not shared with container host
  - It is immutable – cannot be modified
  - You add layers (which are basically filesystem changes) on top to create a final image
- Base OS layer consists of two separate images…
  - Base Layer – All functionality
  - Update Layer – Patches/Updated files





FROM golang:1.9.2-alpine3.6 AS build # Install tools required for project # Run `docker build --no-cache .` to update dependencies RUN apk add --no-cache git RUN go get github.com/golang/dep/cmd/dep # List projec

Base Images for Windows

- Provides a snapshot of the of an OS file system – not the full OS
- It is immutable – cannot be modified

- Windows Server Core…
  - Minimal installation of Windows Server 2016
  - Contains only core OS features
  - Command-line access only
  - https://hub.docker.com/r/microsoft/windowsservercore/

- Nano Server…
  - Available only as container base OS image (no VM support)
  - 20 times smaller than Server Core
  - Headless – no logon or GUI
  - Optimized for .NET Core applications
  - https://hub.docker.com/r/microsoft/nanoserver/

Dockerfile

- Text file with Docker commands – instructions needed to construct an image

```
# Pulls Dotnet Image from Docker Hub
FROM microsoft/dotnet:2.0.0-sdk-2.0.2-nanoserver
# Creates a Working Directory
WORKDIR dockerdemo
#Add all the required applications artifacts inside the container
ADD . .
#Command to Run when container is up and running
ENTRYPOINT ["dotnet","bin/Debug/netcoreapp2.0/publish/Catalog.API.dll"]
# Exposes the Container Port
EXPOSE 8082
# Exposes environment variable
ENV ASPNETCORE_URLS http://0.0.0.0:8082
```

- The *docker build* command parses the Dockerfile to build a new container image.

```
docker build -t mycoolimage:1.0 .
```

Common Dockerfile Instructions

- **FROM** initializes a new build stage and sets the Base Image for subsequent instructions.
- **LABEL** is a key-value pair, stored as a string. You can specify multiple labels for an object, but each key-value pair must be unique within an object.
- **RUN** will execute any commands in a new layer on top of the current image and commit the results.
- **WORKDIR** instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it.
- **ADD** instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.
- **COPY** instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.
- **CMD** provide defaults for an executing container. These defaults can include an executable.
- **ENTRYPOINT** allows you to configure a container that will run as an executable.
- **EXPOSE** instruction informs Docker that the container listens on the specified network port(s).

Complex Dockerfile

```
FROM golang:1.9.2-alpine3.6 AS build

# Install tools required for project
# Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

Demonstration

Build an Image

```
dotnet publish
docker build . –t mycoolimage:1.0
docker images
```

"If an image is a class, then a container is an instance of a class— a runtime object."

- An ***image*** is immutable & defines a version of a single service with its dependencies (runtimes, etc.)
  - Use the same container image everywhere: Dev, test, staging, production
- A ***container*** runs an image in an isolated environment
  - Multiple containers (services) can run side-by-side within a single PC/VM

Microsoft Confidential

Basic Execution - Docker Run command

- Creates writable container over an image and starts it
- Manages resources allocated to container
  - CPU, memory, networking, port mappings, volume mappings, name
- Can stop and (re)start container without losing changes
  - Docker commit command can save changes
  - Otherwise, changes are lost when container is destroyed

```
docker run –d –p 8090:8082  --name contaniner1 imagedemo:1.0
docker ps
```

- -d flag tells it to run in detached mode (session not attached to container)
- -p flag maps <host port>:<container port>
  - Runs on 8090 on host, but on 8082 in container

Microsoft Confidential

Demonstration

Run Docker Container


```
docker run –d –p 8090:8082  --name contaniner1 imagedemo:1.0
docker ps
```

Docker Inspect Command

- Returns low-level information on a Docker object
- JSON-array of container information
- Use to get container IP address, among other things
- Attributes of interest
  - “Isolation” : “hyperv”
  - IP Address

Microsoft Confidential

Demonstration

Run Docker Inspect
Inspect Isolation level
Get Container IP Address

```
docker inspect <Container ID>
```

Scale Out Docker Instance

- Can manually *scale out* container instance
- Create multiple containers from single image

```
docker run –d –p 8091:8082  --name containner2 imagedemo:1.0
```

```
docker run –d –p 8092:8082  --name containner3 imagedemo:1.0
```

- Can view containers from external host

```
http:// DESKTOP-8539463:8091/swagger
```

```
http:// DESKTOP-8539463:8092/swagger
```

Microsoft Confidential

Start-Up Performance

- Virtual Machine
  - 30 seconds – 1 minute
- Windows Server Containers
  - Nano – 1 second
  - Server Core – 2 seconds
- Hyper-V Containers
  - Nano – 5 seconds
  - Server Core – 10 seconds

Demonstration

- Can stop, restart and finally remove a container

```
docker stop <id> <id> <id>
```

```
docker ps
```

```
docker ps -a
```

```
docker start <Container Id>
```

```
docker rm <id> <id> <id>
```

Run Commands in Container

- Can start container and interactively run invoke commands inside of it
- -it flag enables interactive input into the container

- Run the base image with SDK

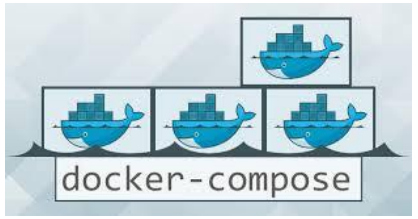
```
docker run –it <Image Id> powershell
type license.txt
```

Microsoft Confident

Managing Multiple Containers

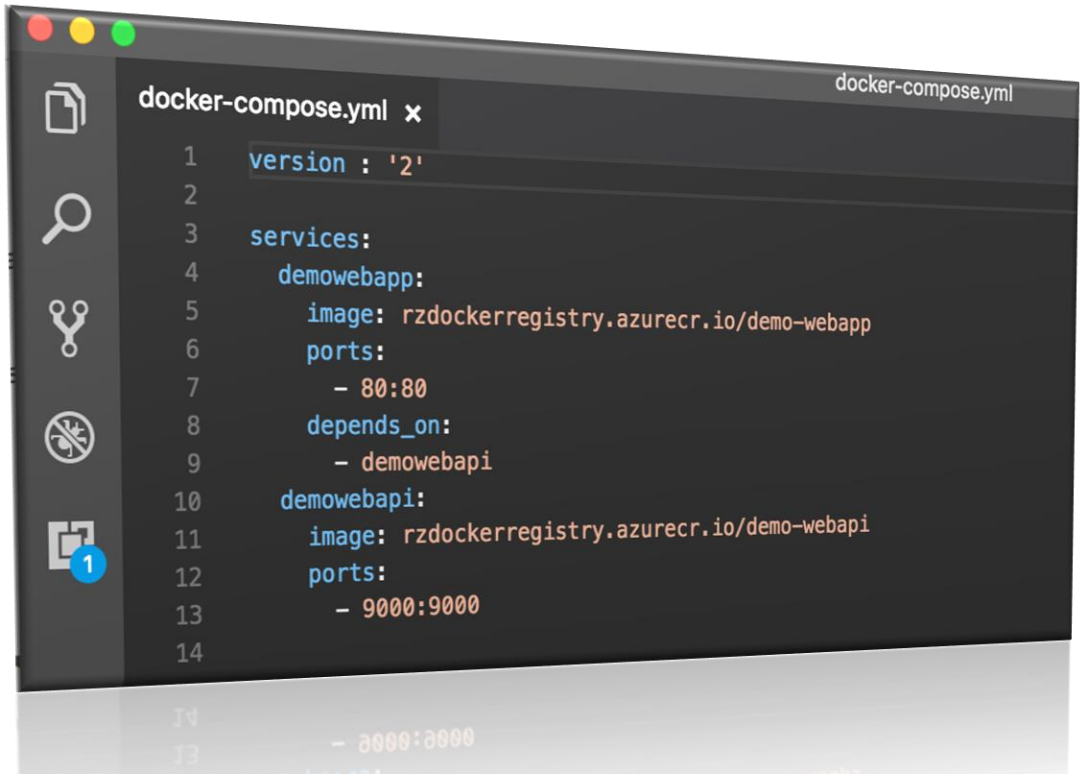
- Running a single container is straight forward
- How do you run several ***different*** containerized microservices simultaneously and enable communication between them?
- How do to manage configurations between environments?

Docker Compose

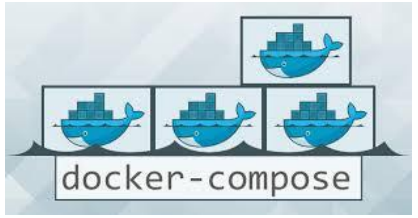


- Compose is a tool for defining and running multi-container Docker applications
- Single command to create and start all the containerized services as a unit
- Single command to stop all the services
- Enables networking stack and service discoverability

- Three-step process:
  - Define each service with a *Dockerfile*
  - Specify the services that will run together in a *docker-compose.yml*
  - Lastly, run *docker-compose up* and to compose each container and start the app



Docker Compose - Declarative



Docker Compose is *declarative*, meaning that you simply describe the *desired state* for an application with a configuration file and the Docker engine figures out exactly how to make it happen

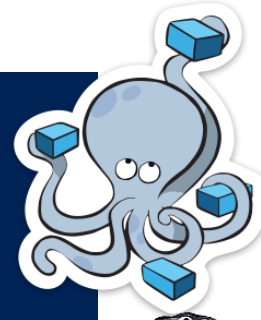
```
services:

  musicstore:
    image: musicstore:1.0
    build:
      context: ./UI
      dockerfile: Dockerfile
    environment:
      - ApiGateway=apigateway.api:8084
    depends_on:
      - apigateway.api

  apigateway.api:
    image: apigateway:1.0
    build:
      context: ./ApiGateway
      dockerfile: Dockerfile
    depends_on:
      - catalog.api
      - basket.api
      - ordering.api
    environment:
      - "ServiceUrl:Catalog=http://catalog.api:8082“/
      - "ServiceUrl:Basket=http://basket.api:8083"
      - "ServiceUrl:Ordering=http://ordering.api:8085"

  catalog.api:
    image: catalog:1.0
    build:
      context: ./Catalog.Service
      dockerfile: Dockerfile
    environment:
      - Data:DefaultConnection:CatalogConnectionString=Server=tcp:learn-activateazure.database.windows.net,1433; .........
      - ServiceBusPublisherConnectionString=Endpoint=sb://learn-activateazure ............
```

- Each container becomes a service
- Compose builds image named “musicstore:1.0”
- Looks for Dockerfile in specified relative directory (called the “context”)
- Accepts environment variables
- Waits for apigateway container to instantiate

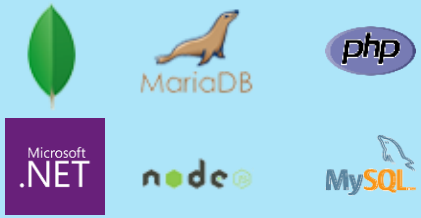
Compose file

- Describes what deployment should look like once deployed – the desired state

Docker compose



Registry



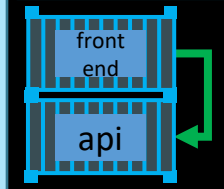
Docker-compose.yml

```
version: '2'
services:
  multiservice:
    image:
      - multiservice:latest
    environment:
      - CustomerAPIService=http://webapi/api/Custome
    ports:
      - "80:80"
    depends_on:
      - webapi
  webapi:
    image:
      - multiserviceapi:latest
```

DOCKER_HOST

Docker daemon

Containers



Images

Multi Service

Multi Service API

Private Registry

Multi Service

Multi Service API

Client

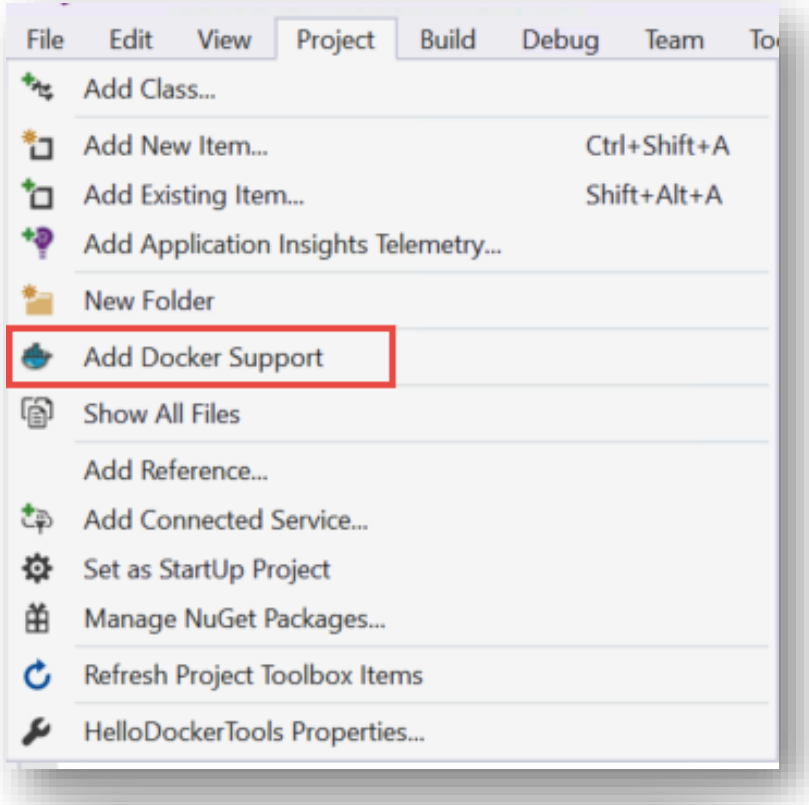
```
Docker-compose up
```

```
Docker-compose down
```

Microsoft Confidential

Visual Studio Tools for Docker

- Microsoft Visual Studio 2017 provides integrated developer experience with Docker
- Build, debug and running .NET Framework and .NET Core web and console applications using Windows and Linux containers.



Multi-Stage Build

- New feature available in Docker 17.05+
- Include multiple FROM statements in single Dockerfile
- Each FROM instruction uses different base, and each begins a new stage of the build
- Selectively copy artifacts from one stage to another, leaving behind everything items not needed in the final image

Multi-Stage Build

```
FROM microsoft/dotnet:2.1-aspnetcore-runtime-nanoserver-1709 AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/dotnet:2.1-sdk-nanoserver-1709 AS build
WORKDIR /src
COPY WebApplication3/WebApplication3.csproj WebApplication3/
RUN dotnet restore WebApplication3/WebApplication3.csproj
COPY . .
WORKDIR /src/WebApplication3
RUN dotnet build WebApplication3.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish WebApplication3.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "WebApplication3.dll"]
```

Targets Core runtime base image

Targets Core SDK base image

Copies source and prj file to src directory

Performs Restore and Build

Performs a Publish

Targets Core runtime base image

Demonstration

Visual Studio Tools for Docker

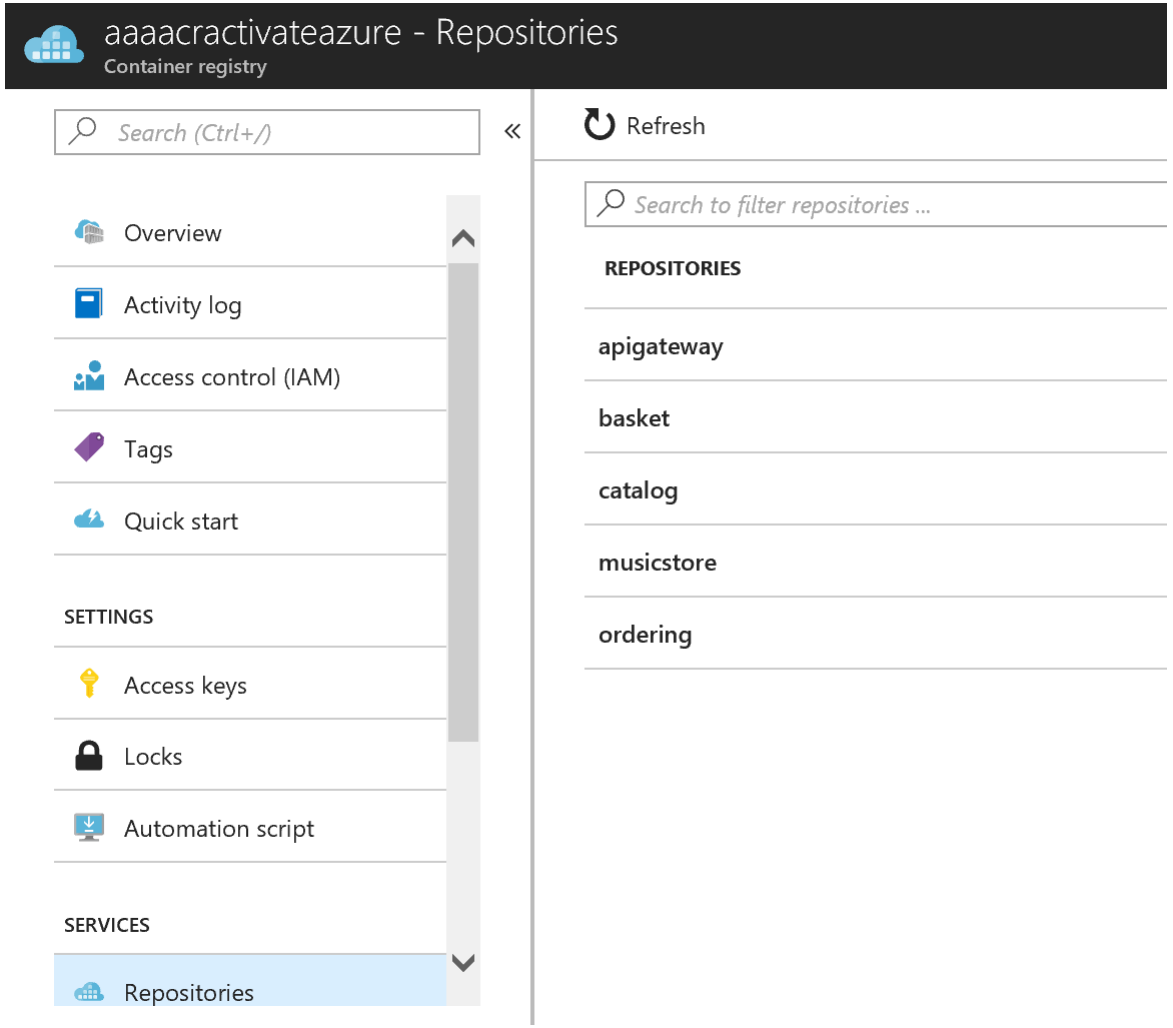
Docker Run from Visual Studio

- Docker Run command when starting from Visual Studio 2017

```
docker run -dt -v "C:\Users\Admin\onecoremsvsmon\15.0.28010.2016:C:\remote_debugger:ro" -v "C:\Users\Admin\Documents\Visual Studio 2017\Projects\CoreWithContainers:C:\app" -v "C:\Users\Admin\.nuget\packages\:c:\.nuget\fallbackpackages2" -v "C:\Program Files\dotnet\sdk\NuGetFallbackFolder:c:\.nuget\fallbackpackages" -e "DOTNET_USE_POLLING_FILE_WATCHER=1" -e "ASPNETCORE_ENVIRONMENT=Development" -e "NUGET_PACKAGES=c:\.nuget\fallbackpackages2" -e "NUGET_FALLBACK_PACKAGES=c:\.nuget\fallbackpackages;c:\.nuget\fallbackpackages2" -P --entrypoint C:\remote_debugger\x64\msvsmon.exe corewithcontainers:dev /noauth /anyuser /silent /nostatus /noclrwarn /nosecuritywarn /nofirewallwarn /nowowwarn /fallbackloadremotemanagedpdbs /timeout:2147483646
```

Docker Registry

- Repository to store docker images
- Searchable
- Public Registry – Hub.Docker.com
- Private Registries – Instanced for you
  - Can be hosted in Docker, Azure, AWS, Google, …
- Must tag image with repository name prefix

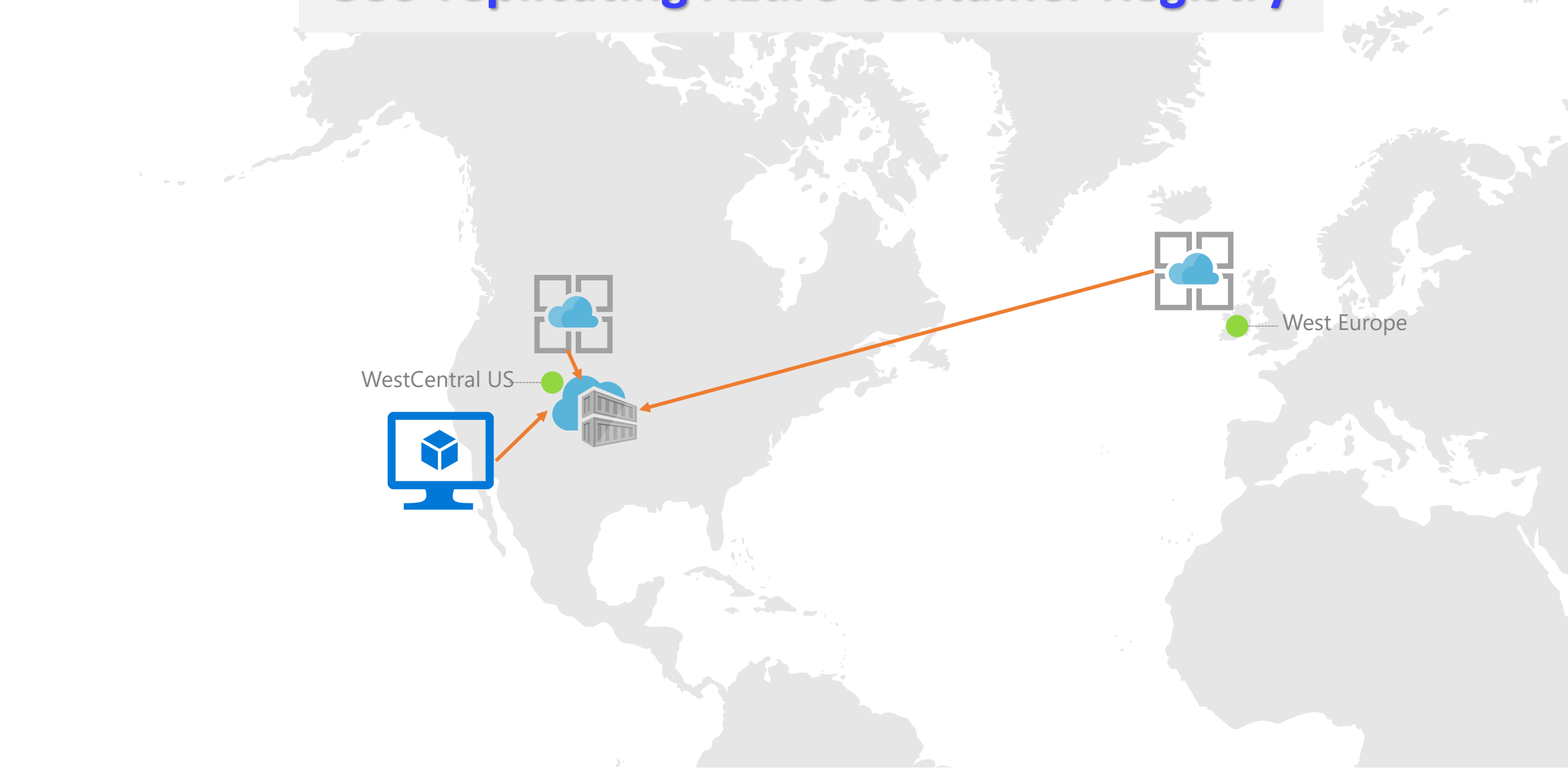


Demonstration

Push Image to Azure Container Registry

```
docker login <login sever>
docker images
docker tag <image id> <login server>/imagedemo:1.0
docker images
docker push <login ACR Name>/imagedemo:1.0
```

Geo-replicating Azure Container Registry



Geo-replication

- Regions are multi-master
  - Each Docker push creates new image layers, eliminating normal update conflicts
  - Updating an existing tag; last writer wins, and will be eventually consistent across all regions
  - Push can be done to any region, replicated to others
- RBAC is shared across all regions
- Each region has geo-redundant storage

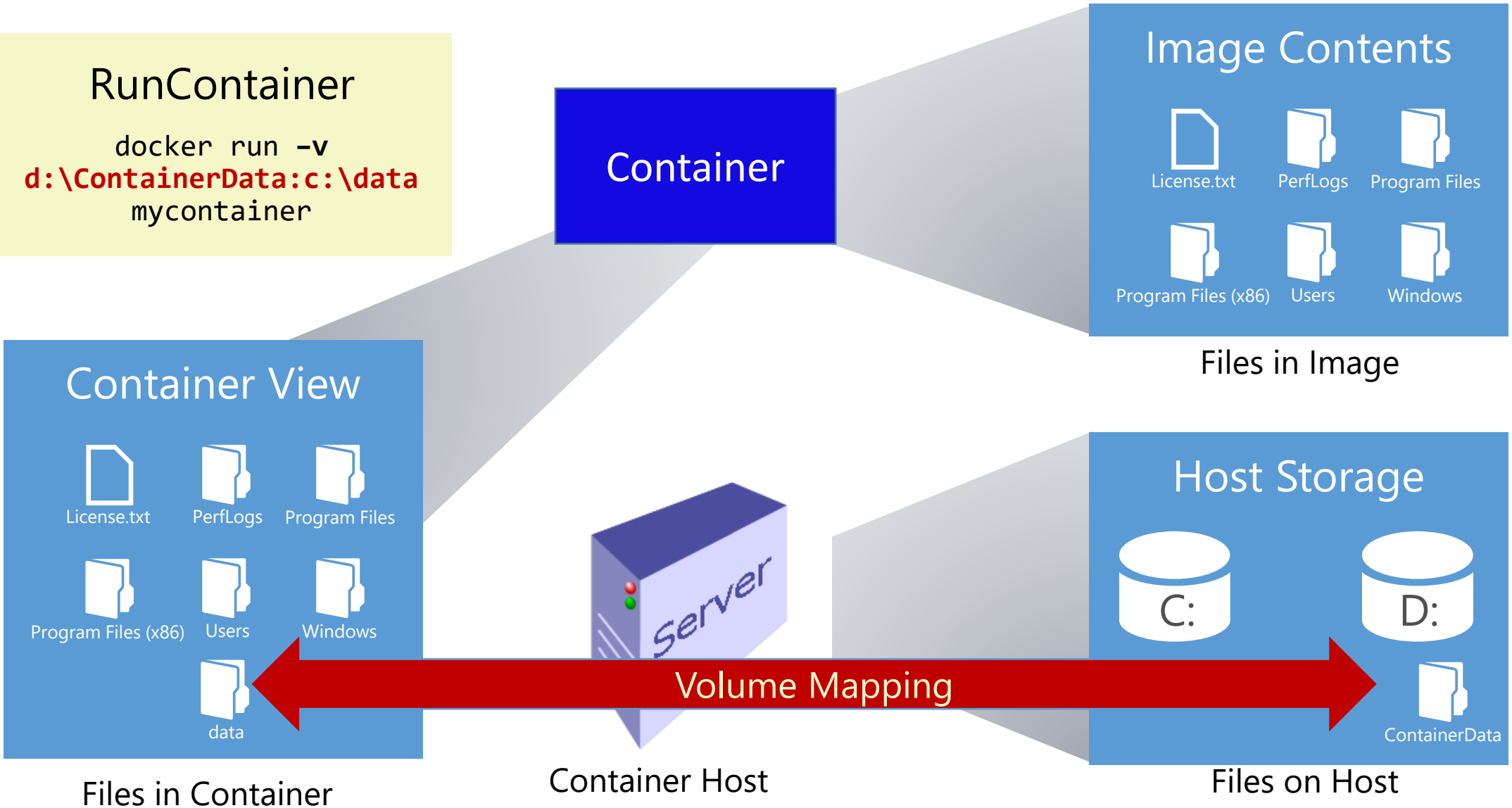
Persistent Storage

- Docker containers are immutable
- To save data, you must provide external storage
- Docker volumes…
  - Enables storage persistence – on the host machine
  - Enables mapping of storage into containers – using the ‘v’
  - Multiple containers on the same host can access the same location
- Network Storage
  - Containers access SMB shares
  - Accessed though the containers network

Volume Mapping

RunContainer

```
docker run -v
d:\ContainerData:c:\data
mycontainer
```

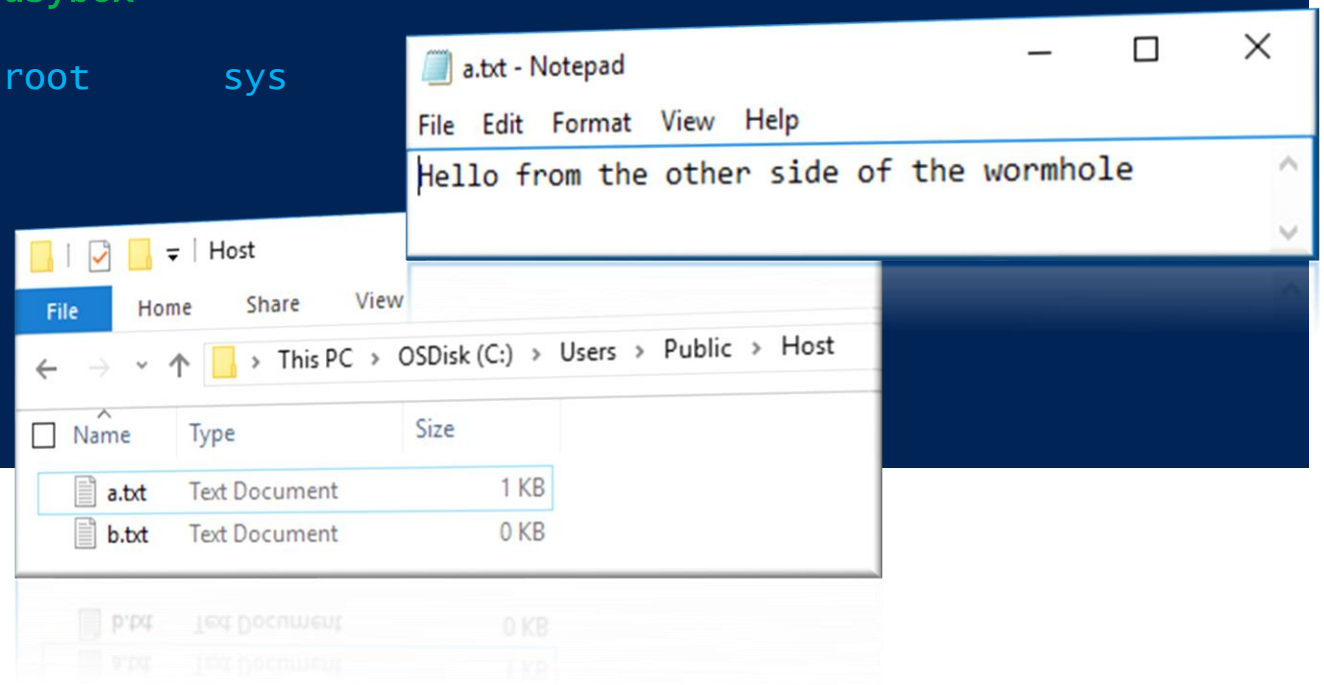


Microsoft Confidential

Docker Volume Mapping

Docker run **-v** hostpath:/containerpath

```
PS docker run -it -v /c/Users/SteveLas:/wormhole busybox
/ # ls
bin       dev       etc       home      proc      root      sys
/ # cd wormhole
/wormhole # touch a.txt
/wormhole # touch b.txt
/wormhole # ls
a.txt  b.txt
/wormhole # cat a.txt
Hello from the other side of the wormhole
/wormhole #
```



Demonstration: Map Volume

Map Volume to Container

docker run –it –v <*volume name on host*>:<*directory on container*> <image Id>

```
docker run –it –v my-cool-volume:c:\mydata <image>
```

Add File to Container Volume

```
Copy con mytextfile.txt
```

Add Content
Show file on host

```
C:\ProgramData\Docker\volumes
```

Windows Containers: Two Flavors

Hyper – V Containers	Windows Containers
Multiple container instances run concurrently on host	Multiple container instances run concurrently on host
Each container runs inside of a **special** virtual machine	Process isolation provided through namespace, resource control, and process isolation technologies
This provides *kernel level isolation* between each Hyper-V container and the container host	Windows Server containers *share the same kernel* with the host, as well as each other
docker run -it --isolation=hyperv microsoft/nanoserver cmd	docker run -it microsoft/nanoserver cmd

Microsoft Confidential

Microsoft Confidential