Microsoft

Developing Applications with Containers

Microsoft Services

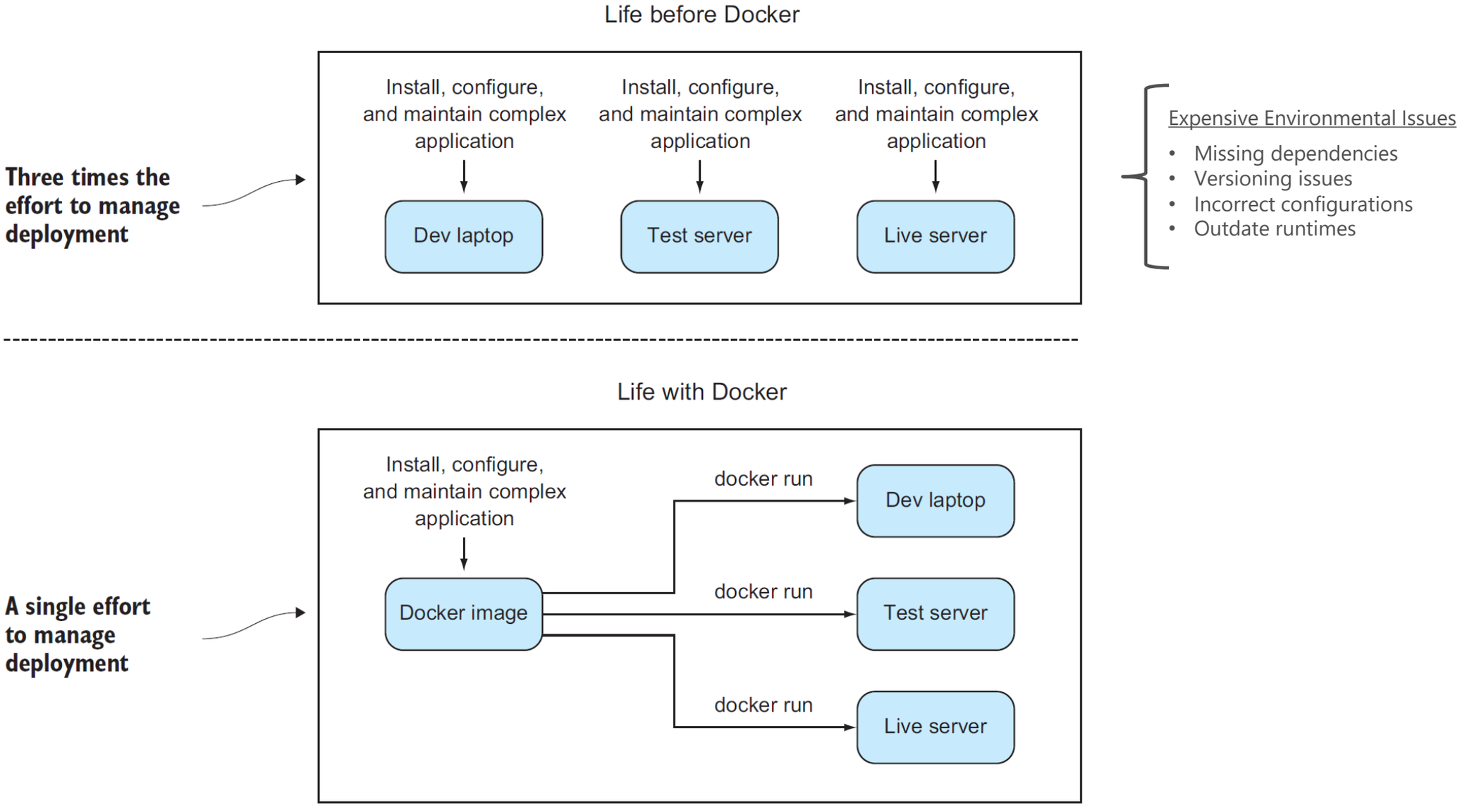
Microsoft

Module 3 – Advanced Docker Topics

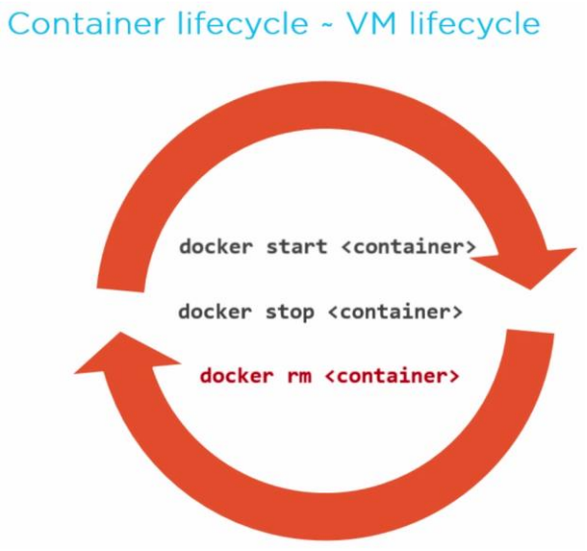
Microsoft Services

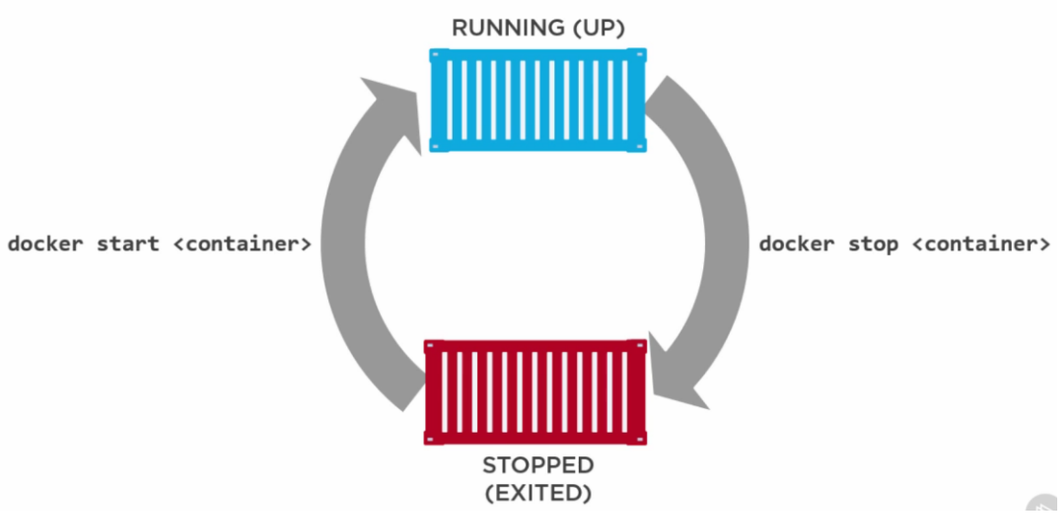
Agenda

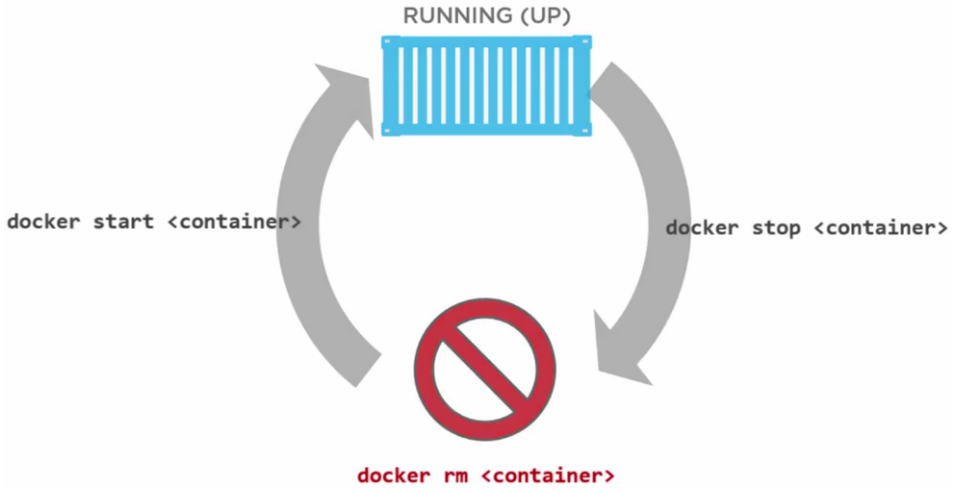
- Docker container lifecycle
- Docker Private Registry
- Multistage Dockerfiles
- Data Management With Docker
- Docker Compose
- Limiting Memory and CPU for Docker containers
- Docker Networking


Life before Docker
Three times the effort to manage deployment
Install, configure, and maintain complex application
Install, configure, and maintain complex application
Install, configure, and maintain complex application
Dev laptop
Test server
Live server
Expensive Environmental Issues
Missing dependencies
Versioning issues
Incorrect configurations
Outdate runtimes
Life with Docker
A single effort to manage deployment
Install, configure, and maintain complex application
Docker image
docker run
Dev laptop
docker run
Test server
docker run
Live server

Container Lifecycle



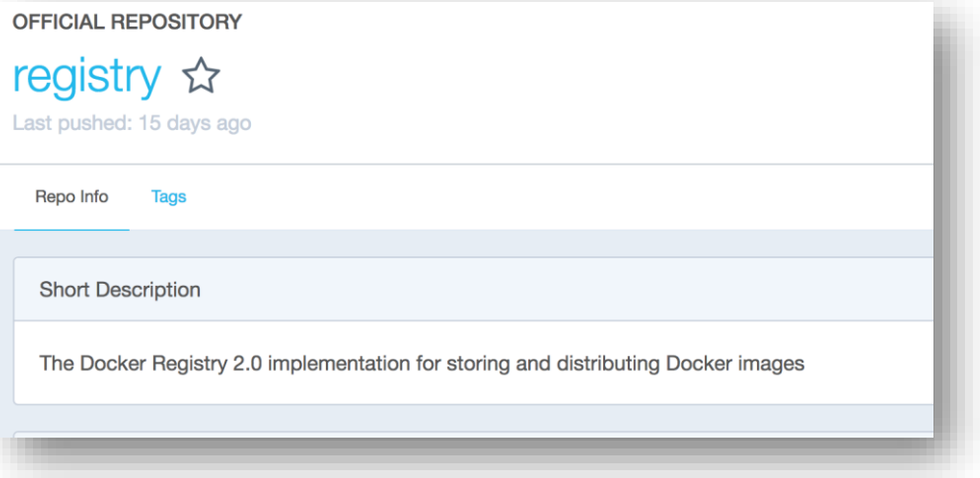




Docker Registry

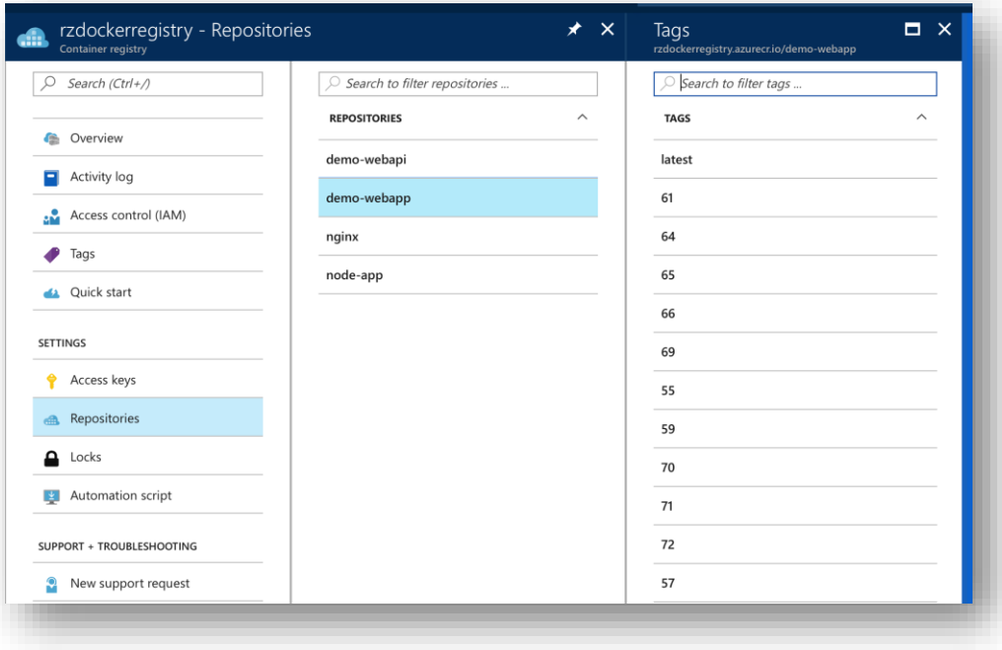
- Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images.

- Usage Pattern:
  - Tightly control where your images are being stored
  - Fully own your images distribution pipeline
  - Integrate image storage and distribution tightly into your in-house development workflow
  - Public (DockerHub) / Private



Docker Private Registry

- Private registry provides better security over public registry (e.g. Docker Hub)
- Azure supports hosting private registry with fine grain Role Based Access Control for management
- Azure private registry can be geo-redundant making it faster to download/upload images based on client location.



Demonstration: *Working with Docker Registry*

Azure Container Registry (ACR)

Push a Custom Image to Private Registry

Pre-Multistage Dockerfile

1. Dockerfile.{purpose} to use for development
2. Dockerfile: Production-centric, containing the app what is needed to run it

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN go get -d -v golang.org/x/net/html \
  && CGO_ENABLED=0 GOOS=linux go build -a
```

```
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY app .
CMD ["./app"]
```

```
#!/bin/sh
echo Building alexellis2/href-counter:build

docker build --build-arg https_proxy=$https_proxy --build-arg http_proxy=$http_proxy \
    -t alexellis2/href-counter:build . -f Dockerfile.build

docker container create --name extract alexellis2/href-counter:build
docker container cp extract:/go/src/github.com/alexellis/href-counter/app ./app
docker container rm -f extract

echo Building alexellis2/href-counter:latest

docker build --no-cache -t alexellis2/href-counter:latest .
rm ./app
```

Multistage Dockerfile (Docker 17.5)

- Use multiple FROM statements in your Dockerfile
- Each FROM begins a new stage of the build and can use a different base image
- Selectively copy artifacts from one stage to another

```
FROM golang:1.7.3 as builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuff

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href
CMD ["./app"]
```

```
docker build -t alexellis2/href-counter:latest
```

Target A Specific Build Stage

- Debug a specific build stage
- Use a debug stage with all debugging symbols or tools enabled, and a lean production stage
- Use a testing stage in which your app gets populated with test data, but building for production using a different stage which uses real data

```
FROM golang:1.7.3 as builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuff

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href
CMD ["./app"]
```

```
docker build --target builder -t alexellis2/href-counter:latest
```

Demonstration: *Target A Specific Build Stage*

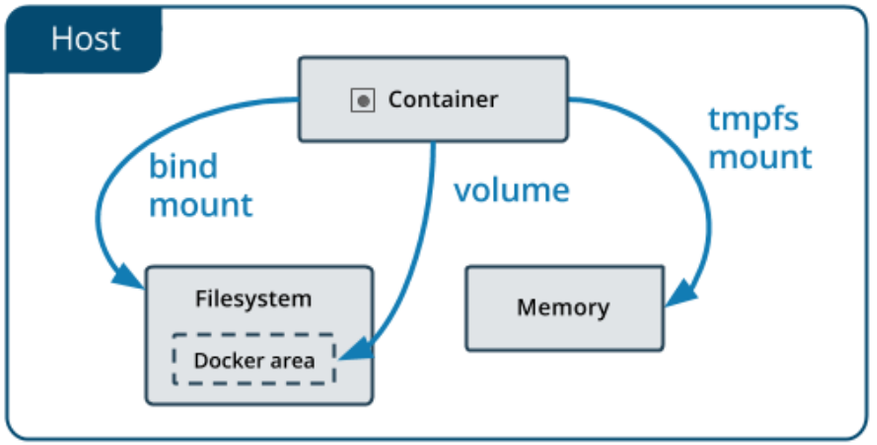
Mount a host directory as a data volume

Managing data in Docker

- Data doesn't persist when a container is removed, and it can be difficult to get the data out of the container if another process needs it
- A container's writable layer is tightly coupled to the host machine where the container is running
- Writing into a container's writable layer requires a storage driver to manage the filesystem

Mounting Data Into A Container From A Host

- TMPFS Mounts: stored in host system memory ONLY (Linux-only)
- Bind Mounts: may be stored anywhere on the host system
- Volumes: stored in a part of the host filesystem which is managed by Docker



Data Volumes Should Be Used Where Possible

- Created explicitly using **`docker volume create`**, or created during container/service creation
- R/W or RO
- Decouple the configuration of the Docker host from the container runtime
- When the mounted container is removed, the volume still exists
- A given volume can be mounted into multiple containers simultaneously
- Support the use of *volume drivers*, allowing data storage on remote hosts or cloud providers

Use Cases For Bind And Tmpfs Mounts

- Sharing config files from the host to the containers
- Sharing source code or build artifacts between dev environment on the Docker host and a container
- File/Directory structure of Docker host is guaranteed to be consistent with the bind mounts required by the containers
- When you do not want the data to persist either on the host machine or within the container

Syntax

- bind mounts and volumes: -v or –volume
- tmpfs mounts: --tmpfs

Docker 17.06

```
$ docker run -d \
  --name devtest \
  -v myvol2:/app \
  nginx:latest
```

- bind mounts, volumes, or tmpfs mounts: --mount

```
$ docker run -d \
  --name devtest \
  --mount source=myvol2,target=/app \
  nginx:latest
```

Sharing Data Among Machines

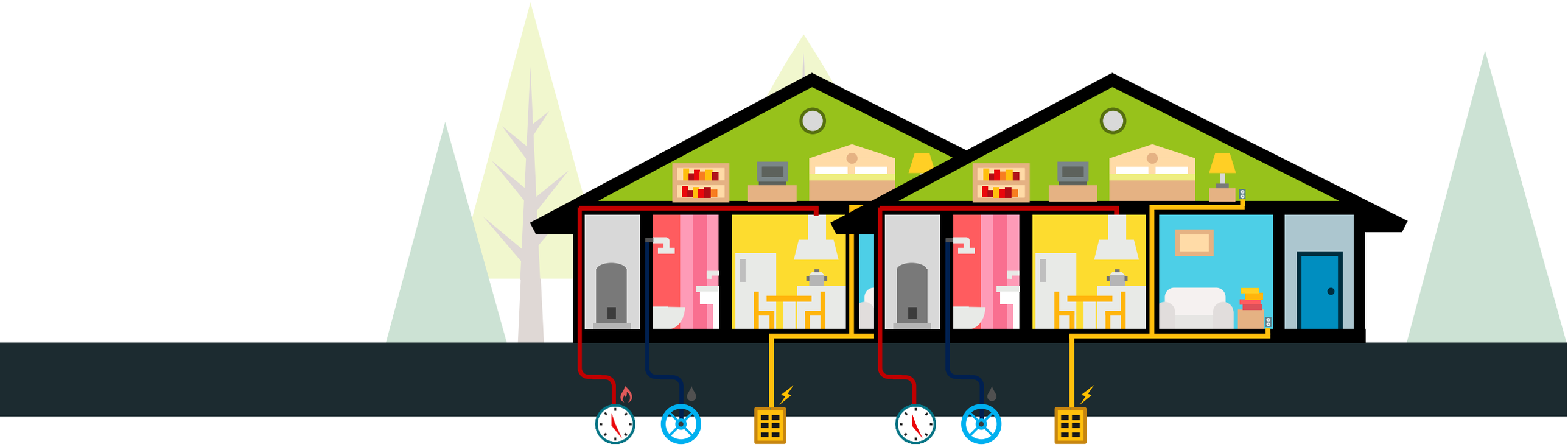
- You might need to configure multiple replicas of the same service to have access to the same files
- When you create a volume using docker volume create, or when you start a container which uses a not-yet-created volume, you can specify a volume driver
- Can use Plug-ins to extend Docker's functionality for mapping shared-volumes

Demonstration: *Working with Data Volumes*

Mount a host directory as a data volume

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



VM vs. Container

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

Limit a Container Resources | Memory

- By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler will allow

- Docker can enforce hard memory limits, which allow the container to use no more than a given amount of user or system memory, or soft limits, which allow the container to use as much memory as it needs unless certain conditions are met, such as when the kernel detects low memory or contention on the host machine.

Option	Description
`-m` or `--memory=`	The maximum amount of memory the container can use. If you set this option, the minimum allowed value is `4m` (4 megabyte).

Limit a Container Resources | CPU

By default, each container's access to the host machine's CPU cycles is unlimited. You can set various constraints to limit a given container's access to the host machine's CPU cycles.

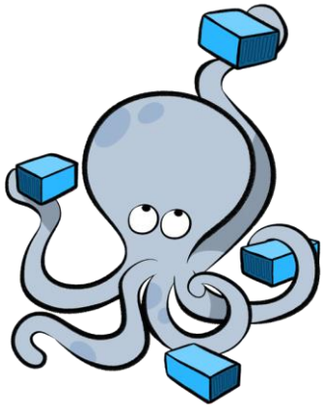
`--cpus=<value>`	Specify how much of the available CPU resources a container can use. For instance, if the host machine has two CPUs and you set `--cpus="1.5"`, the container will be guaranteed to be able to access at most one and a half of the CPUs. This is the equivalent of setting `--cpu-period="100000"` and `--cpu-quota="150000"`. Available in Docker 1.13 and higher.
`--cpu-period=<value>`	Specify the CPU CFS scheduler period, which is used alongside `--cpu-quota`. Defaults to 1 second, expressed in micro-seconds. Most users do not change this from the default. If you use Docker 1.13 or higher, use `--cpus` instead.
`--cpu-quota=<value>`	Impose a CPU CFS quota on the container. The number of microseconds per `--cpu-period` that the container is guaranteed CPU access. In other words, `cpu-quota / cpu-period`. If you use Docker 1.13 or higher, use `--cpus` instead.
`--cpuset-cpus`	Limit the specific CPUs or cores a container can use. A comma-separated list or hyphen-separated range of CPUs a container can use, if you have more than one CPU. The first CPU is numbered 0. A valid value might be `0-3` (to use the first, second, third, and fourth CPU) or `1,3` (to use the second and fourth CPU).

Demonstration: *Limit Container Resources*

Launch container with memory and CPU constraints

Docker Compose

- Compose is a tool for defining and running multi-container Docker applications
- Single compose file defined in .yml/.yaml format defines your application services
- Good for development environments, automated testing environments and single host deployments



Using Docker Compose

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment
3. Run **`docker-compose up`** and Compose will start and run your entire app

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2-stretch-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:2.2-stretch AS build
WORKDIR /src
COPY ["WebAPI/WebAPI.csproj", "WebAPI/"]
RUN dotnet restore "WebAPI/WebAPI.csproj"
COPY . .
WORKDIR "/src/WebAPI"
RUN dotnet build "WebAPI.csproj" -c Release -o /app

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

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

Dockerfile | webapi

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2-stretch-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:2.2-stretch AS build
WORKDIR /src
COPY ["WebFrontEnd/WebFrontEnd.csproj", "WebFrontEnd/"]
RUN dotnet restore "WebFrontEnd/WebFrontEnd.csproj"
COPY . .
WORKDIR "/src/WebFrontEnd"
RUN dotnet build "WebFrontEnd.csproj" -c Release -o /app

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

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

Dockerfile | webapp

```
version: '3.4'

services:
  webapi:
    image: ${DOCKER_REGISTRY-}webapi
    build:
      context: .
      dockerfile: WebAPI/Dockerfile

  webfrontend:
    image: ${DOCKER_REGISTRY-}webfrontend
    build:
      context: .
      dockerfile: WebFrontEnd/Dockerfile
```

Docker-Compose.yaml

```
version: '3.4'

services:
  webapi:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_HTTPS_PORT=44332
    ports:
      - "60287:80"
      - "44332:443"
    volumes:
      - ${APPDATA}/ASP.NET/Https:/root/.aspnet/https:ro
      - ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro
  webfrontend:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_HTTPS_PORT=44362
    ports:
      - "60029:80"
      - "44362:443"
    volumes:
      - ${APPDATA}/ASP.NET/Https:/root/.aspnet/https:ro
      - ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro
```

Docker-Compose-Override.yaml

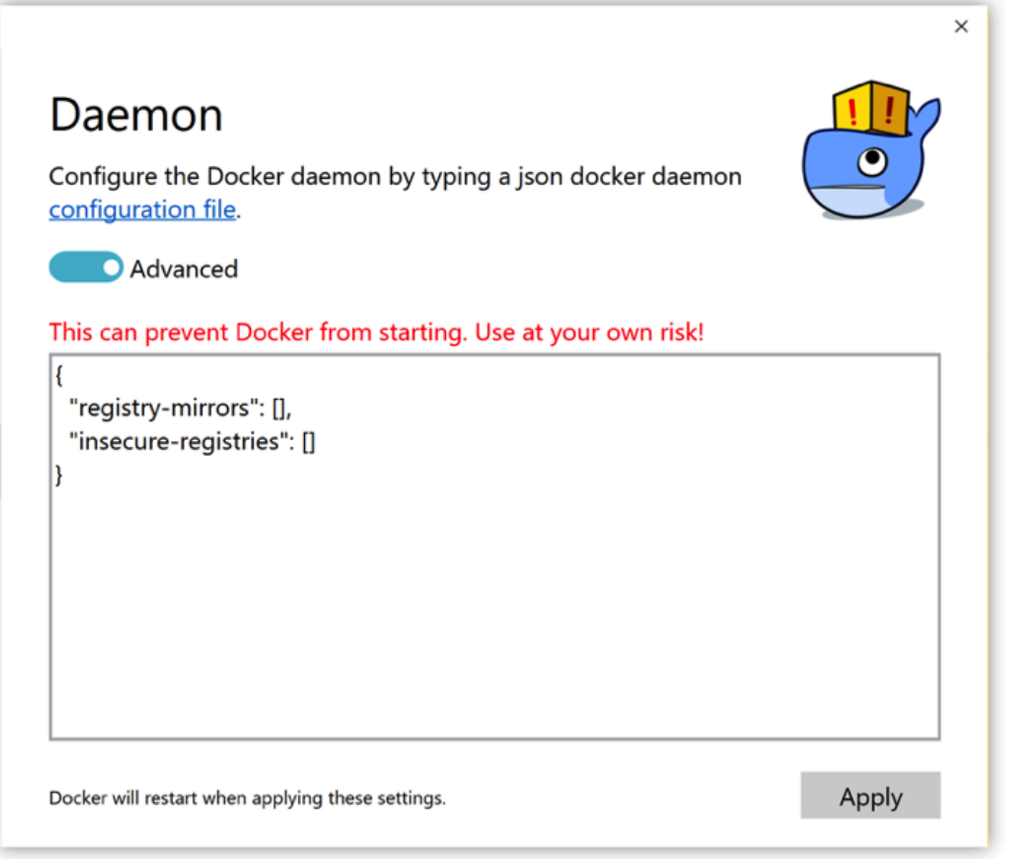
Demonstration: *Docker Compose*

Launch Multi-Container Application using Docker Compose

Configuring Docker with Configuration File

The preferred method for configuring the Docker Engine on Windows is using configuration file. The configuration file can be found at c:\ProgramData\docker\config\daemon.json. If this file doesn't already exist, it can be created.

```
{
    "authorization-plugins": [],
    "dns": [],
    "dns-opts": [],
    "dns-search": [],
    "exec-opts": [],
    "storage-driver": "",
    "storage-opts": [],
    "labels": [],
    "log-driver": "",
    "mtu": 0,
    "pidfile": "",
    "graph": "",
    "cluster-store": "",
    "cluster-advertise": "",
    "debug": true,
    "hosts": [],
    "log-level": "",
    "tlsverify": true,
    "tlscacert": "",
    "tlscert": "",
    "tlskey": "",
    "group": "",
    "default-ulimits": {},
    "bridge": "",
    "fixed-cidr": "",
    "raw-logs": false,
    "registry-mirrors": [],
    "insecure-registries": [],
    "disable-legacy-registry": false
}
```



Network Drivers- Linux containers

- Bridge
  - Default network driver
  - Usually used when your applications run standalone containers that need to communicate
- Host
  - For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly
  - Available for swarm services on Docker 17.06 and higher
- Overlay
  - Connects multiple Docker daemons together and enables swarm services to communicate with each other
  - Swarm service to standalone container or between two standalone containers on different docker daemons
  - Removes need to do OS-level routing between these containers
- Macvlan
  - Assign MAC address to a container, so it will appear as a physical device on your network
  - Best choice when dealing with legacy apps that expect to be directly connected to the physical network vs. routed through the Docker host's network stack
- None
  - Usually used in conjunction with a custom network driver
- Network plugins
  - Install and use third-party plugins

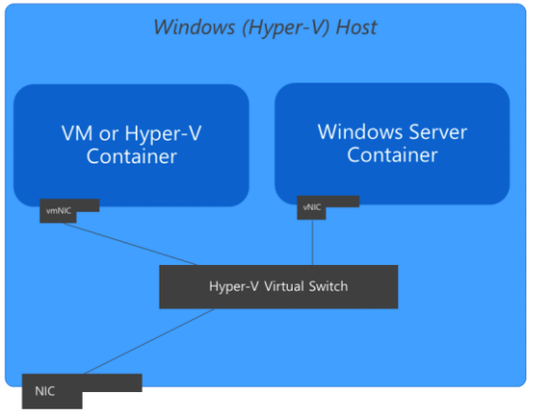
Docker Networking

- Windows containers support 5 different networking drivers or modes: *nat, overlay transparent, l2bridge, and l2tunnel*

- Depending on your physical network infrastructure and *single vs multi-host networking requirements*, you should choose the network mode which best suits your needs

- The docker engine creates a *NAT network* by default when the docker service first runs
  Default *internal IP prefix* created is *172.16.0.0/16*
  Container endpoints will be automatically attached to this *default network* and assigned an *IP address from its internal prefix*

- If your container host IP is in this same prefix, you will need to change the NAT internal IP prefix



Windows Container Network Drivers

- NAT – containers attached to a network created with the 'nat' driver will receive an IP address from the user-specified (--subnet) IP prefix. Port forwarding / mapping from the container host to container endpoints is supported.

- Transparent – containers attached to a network created with the 'transparent' driver will be directly connected to the physical network. IPs from the physical network can be assigned statically (requires user-specified --subnet option) or dynamically using an external DHCP server.

```
docker network create -d <NETWORK DRIVER TYPE> <NAME>
```

Windows Container Network Drivers (Cont.)

- Overlay - when the docker engine is running in swarm mode, containers attached to an overlay network can communicate with other containers attached to the same network across multiple container hosts. Each overlay network that is created on a Swarm cluster is created with its own IP subnet, defined by a private IP prefix. The overlay network driver uses VXLAN encapsulation.

- L2bridge - containers attached to a network created with the 'l2bridge' driver will be in the same IP subnet as the container host. The IP addresses must be assigned statically from the same prefix as the container host. All container endpoints on the host will have the same MAC address due to Layer-2 address translation (MAC re-write) operation on ingress and egress.

- L2tunnel - *this driver should only be used in a Microsoft Cloud Stack*

Demonstration: *Networking*

List all Docker Networks

Create a Custom NAT Network

Run Container on Custom Network