Microsoft

Developing Applications with Containers

Microsoft Services

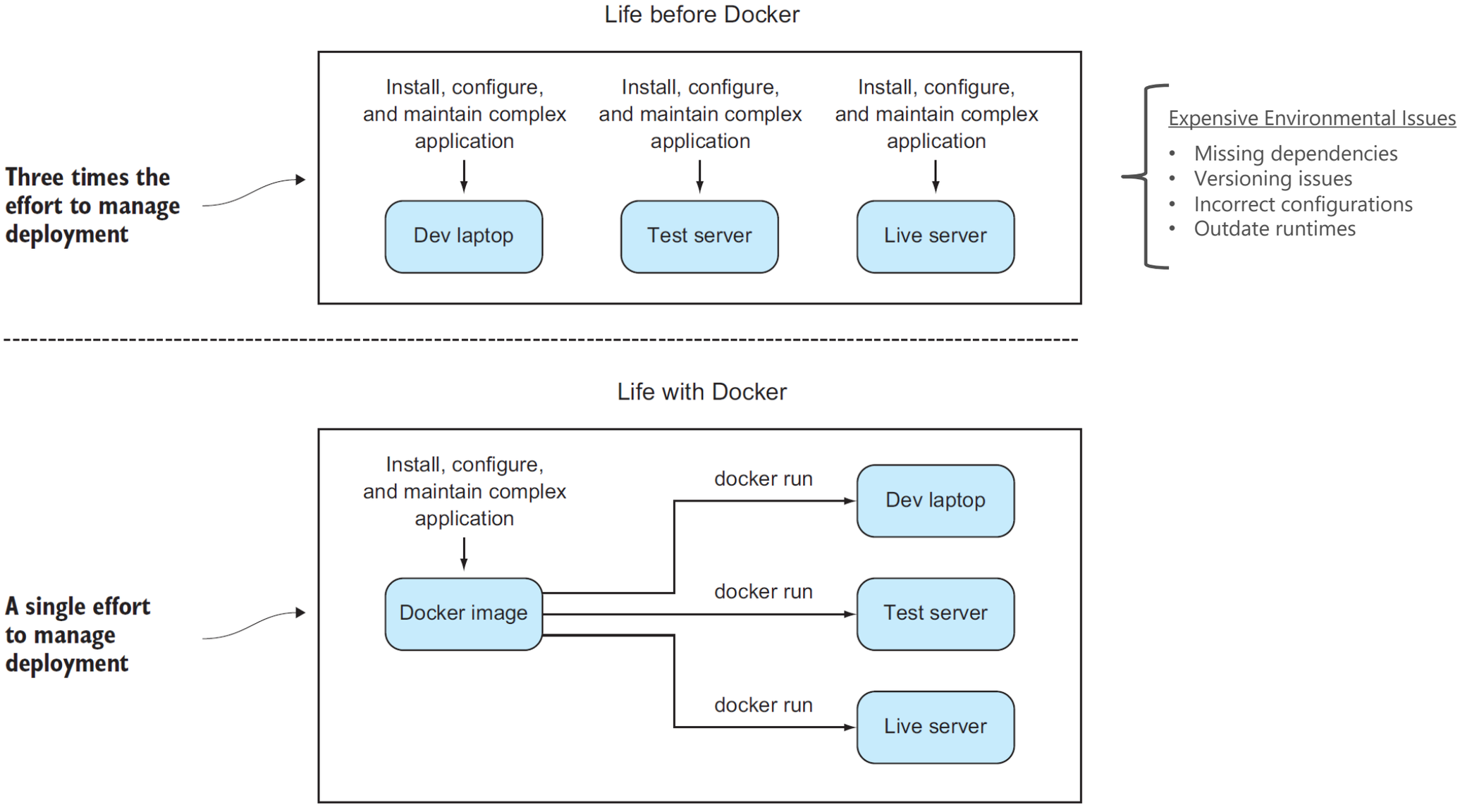
Microsoft

Module 4-1 – Advanced Docker Topics

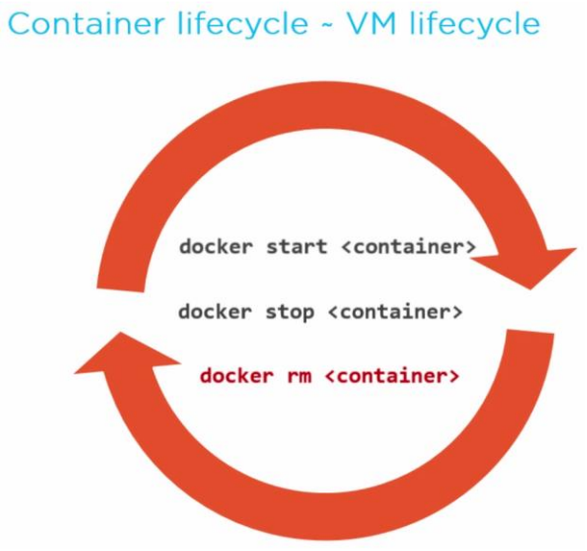
Microsoft Services

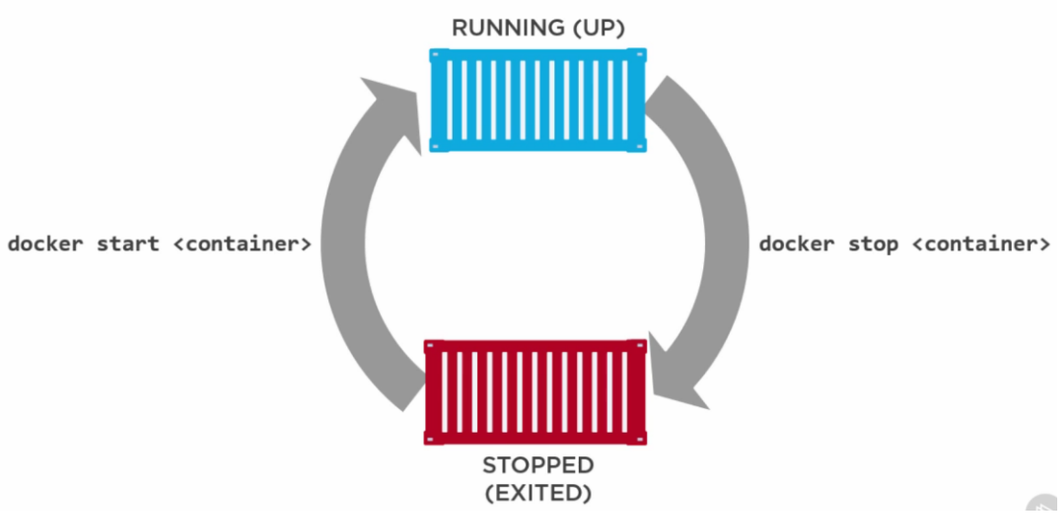
Agenda

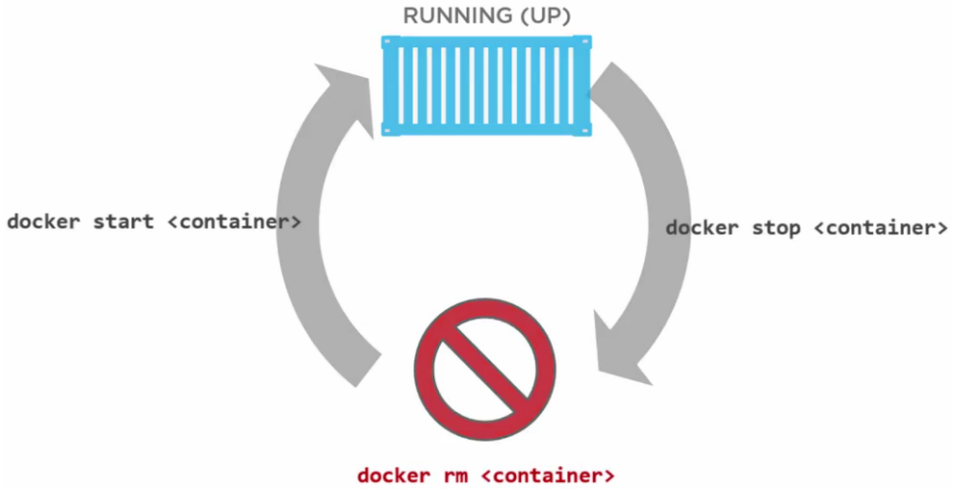
- Docker Container Lifecycle
- Docker Private Registry
- Multistage Dockerfiles
- Data Management With Docker
- Docker Compose


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



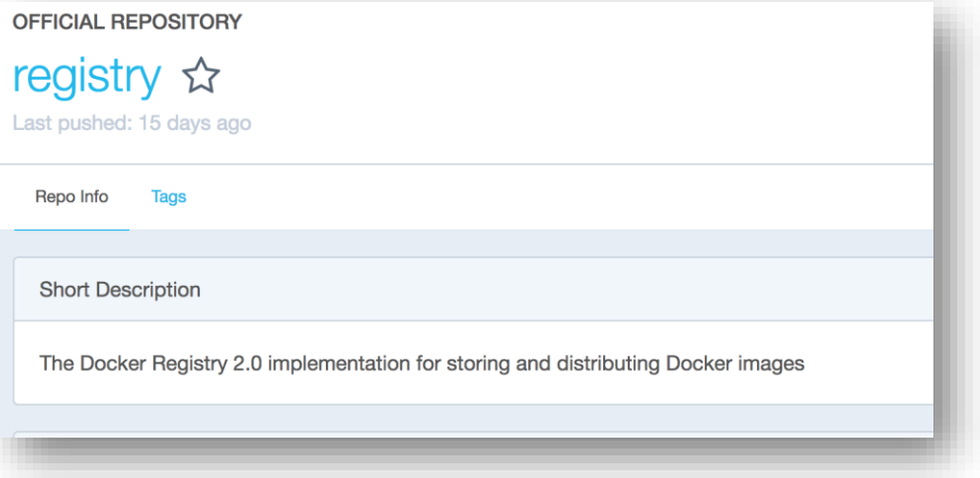




Docker Registry

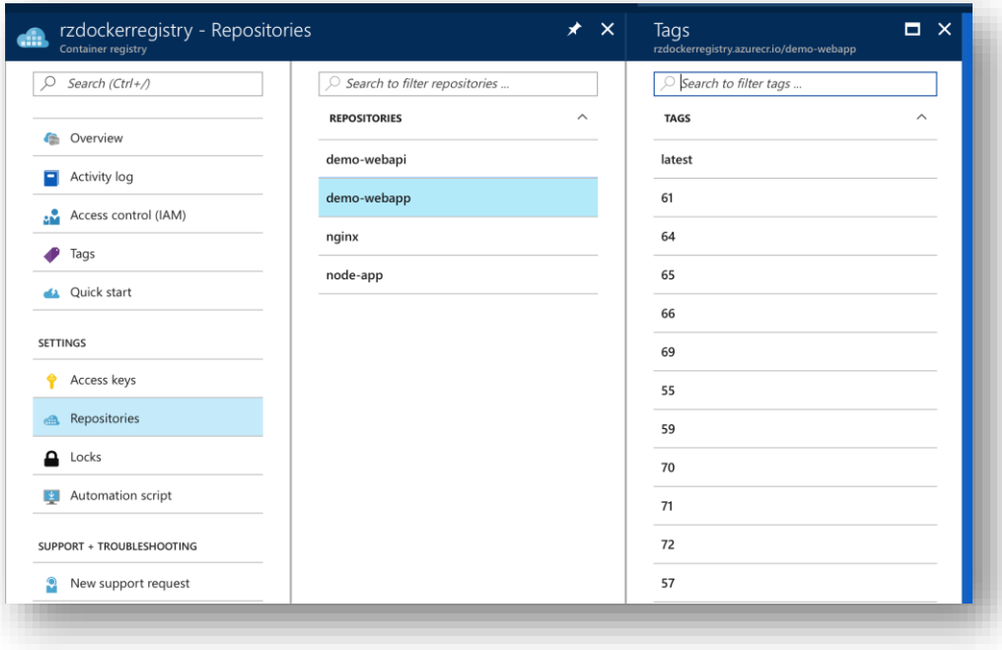
- Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images.

- 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"]
```

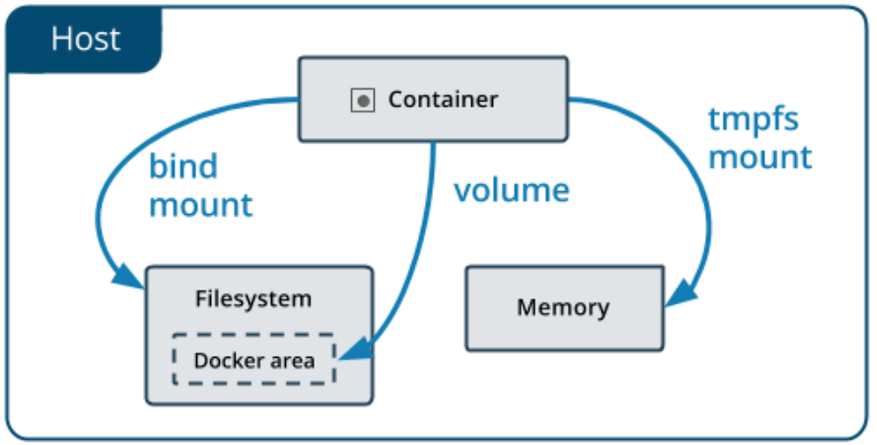
```
docker build -t alexellis2/href-counter:latest
```

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

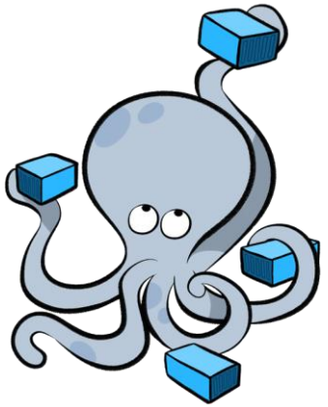
- You might need to configure multiple replicas of the same service to have access to the same files
- When you create a volume using docker volume create, or when you start a container which uses a not-yet-created volume, you can specify a volume driver
- Can use Plug-ins to extend Docker's functionality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

Demonstration: *Working with Data Volumes*

Mount a host directory as a data volume

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: *Limit Docker Compose*

Launch Multi-Container Application using Docker Compose