
DOCKER FOR .NET DEVELOPERS
BRENDON MATHESON

Your Presenter

- Brendon Matheson
- Australian
- 11yr Bangkok Resident

Currently Working On

- Healthcare (Architect at Orion Health)
- Cloud / Multi-Tenant / SaaS
- Functions-as-a-Service (FaaS)

Session Plan

- Basics
  - What is Docker?
  - Docker Basics
  - Serve static content in IIS (nanoserver)
- Real
  - Dockerize a .NET Core WebAPI microservice (nanoserver & linux)
- Production
  - Multi-container solution with docker-compose (linux)
  - Multi-container solution with docker-compose (nanoserver)
  - Scheduling a cluster with Docker Swarm
- Cloud
  - Deploying to AWS ECS
- Dev
  - Continuous Integration
  - Testing Containers
- Tooling Etc
  - Docker Cloud
  - Visual Studio 2017 Support for Docker


WHAT IS DOCKER?

What is Docker?

Packaging, deployment and execution tool

Problems

- Environmental differences
- Complex deployment processes
- Conflicting dependencies

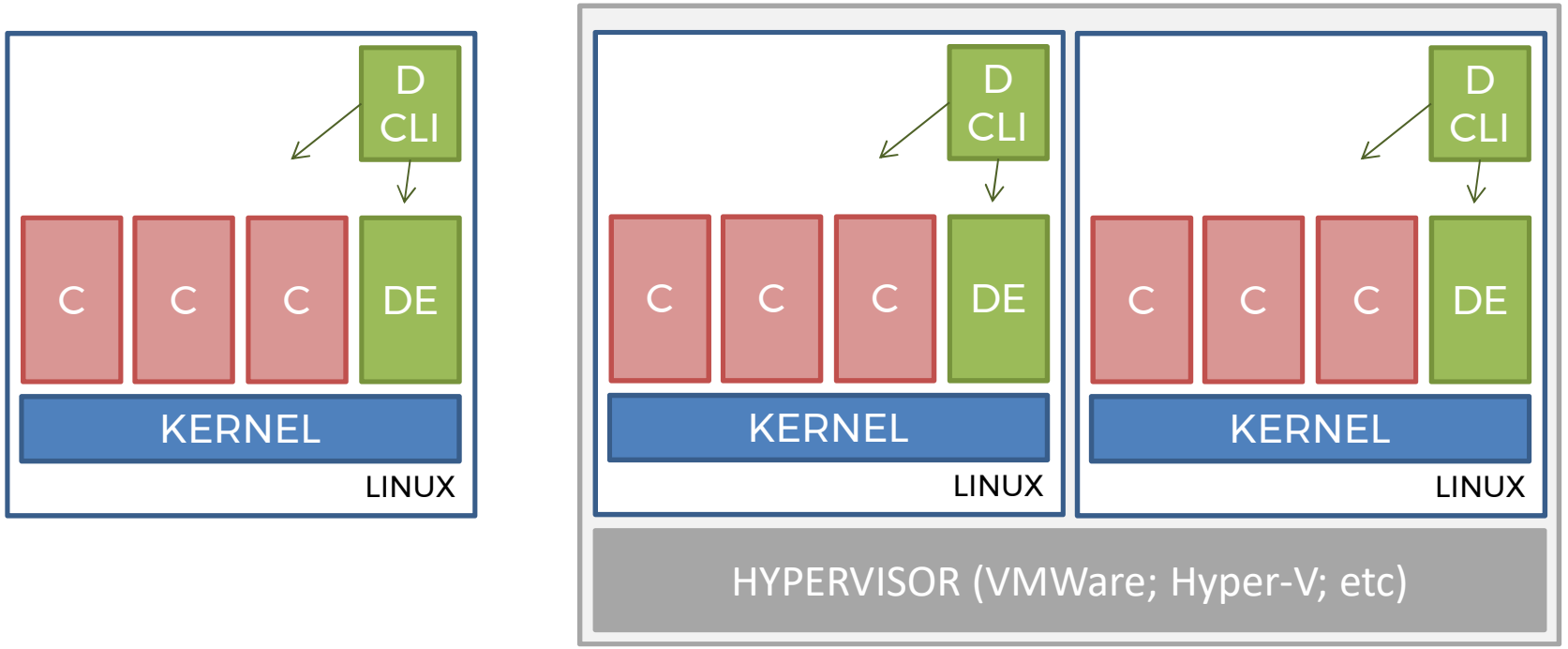
Solution

- Process isolation
- Bundle app and dependencies into containers
- Consistency and portability

TODO

- https://www.datadoghq.com/docker-adoption/

Docker on Linux



Docker on Windows

Docker on Windows – Two Models:

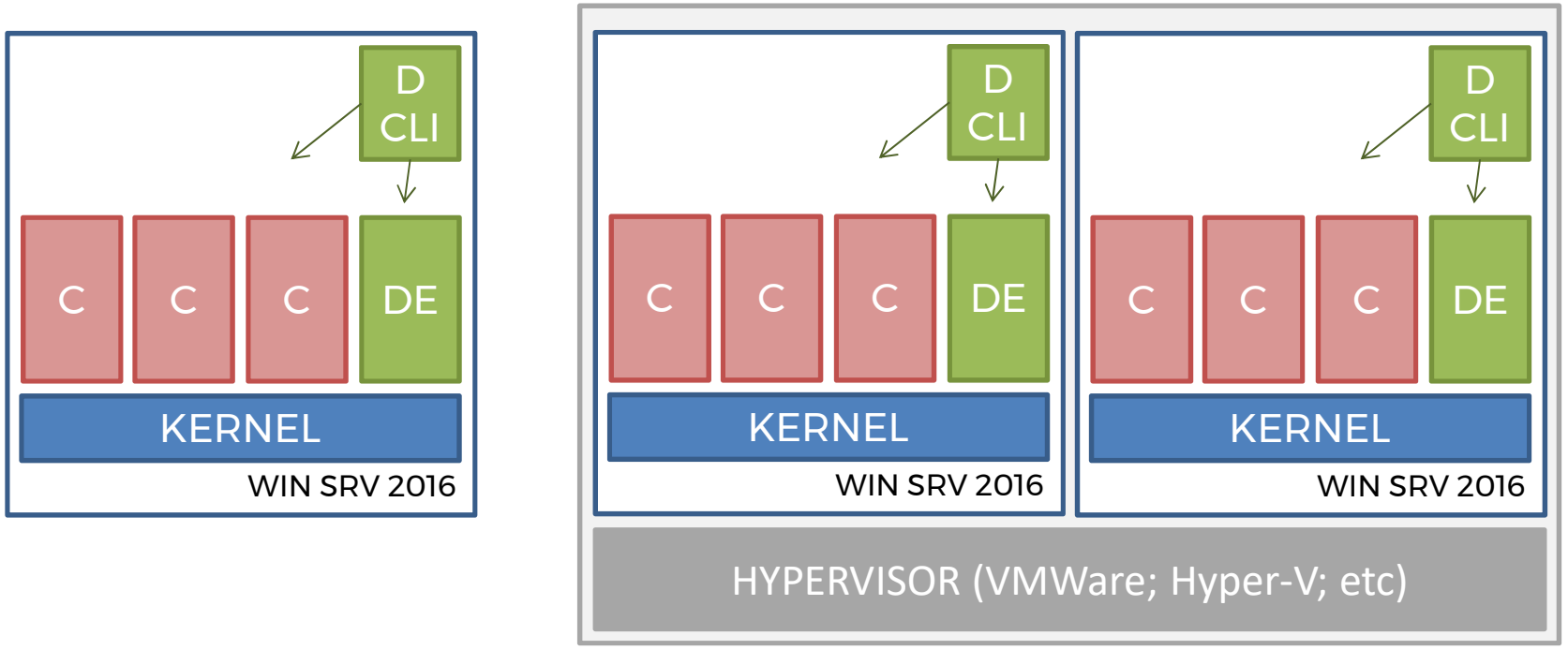
- **Windows Containers** – Kernel-level support like Linux
  - Windows Server 2016
- **Hyper-V Isolation** – Virtualization-based shim
  - Windows Server 2016
  - Windows 10
    - Version 1511 / November 2016 Update / Build 10586

Windows 10

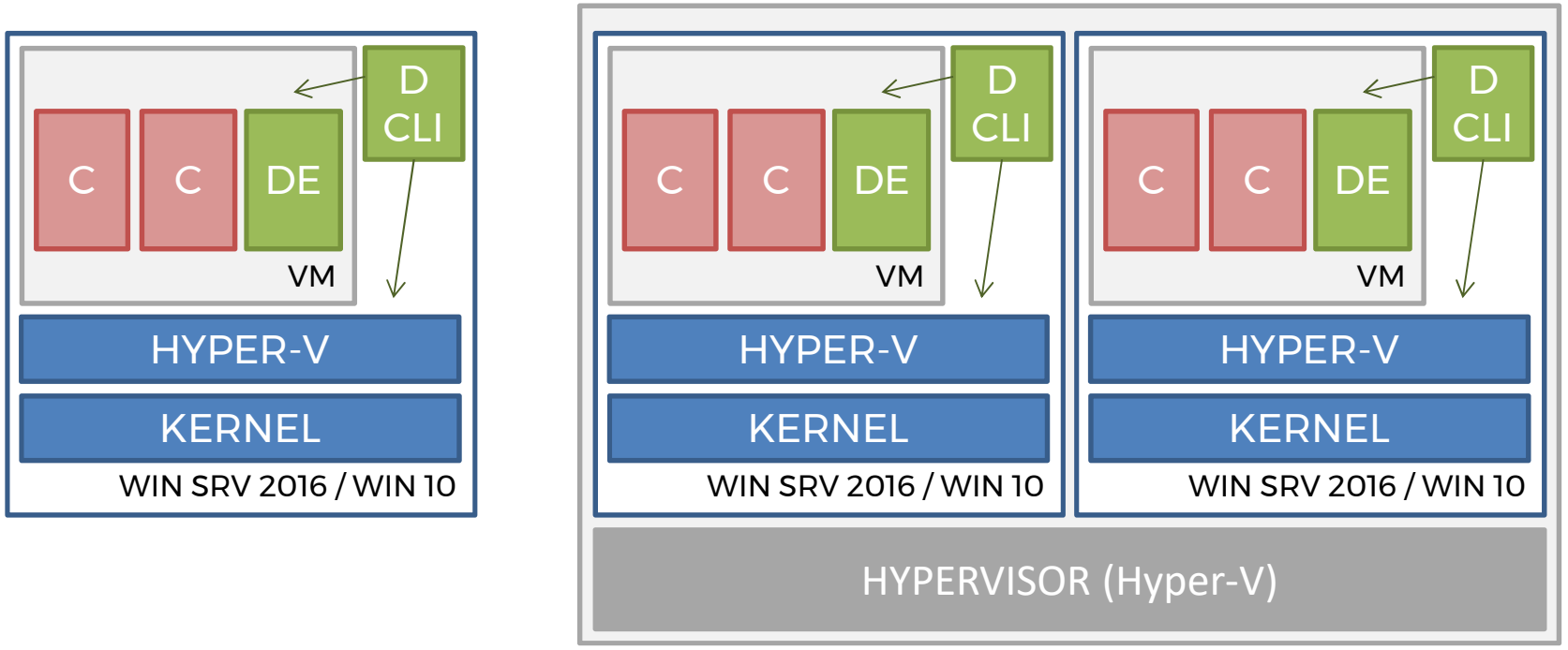
TODO

- https://docs.microsoft.com/en-us/virtualization/windowscontainers/quick-start/quick-start-windows-10

Windows Containers



Hyper-V Isolation



Virtualization vs Containerization

Docker is a cool n virtualization techno y

Virtualization vs Containerization

Virtualization

- Virtual hardware
  - CPU
  - Disk
  - Memory
  - Devices
- Guest OS and software installed into VM

VM's => System-Oriented

Containerization

- Native hardware – no hypervisor
- Allocate resources with control groups (on Linux)
- Host kernel is used by containerized process
- No additional OS install
- Supporting software and libraries are bundled into the container

Containers => Service-Oriented


DOCKER BASICS

hello-world

- Run it!

```
docker run hello-world
```

- Review https://hub.docker.com/_/hello-world/
- Pull

```
docker pull debian:9
```

- Check C:\Users\Public\Documents\Hyper-V\Virtual hard disks

hello-world

- Run an interactive session in Debian 9

```
docker run –i -t debian:9 /bin/bash
```

- Run a detached nginx instance

```
docker run –d nginx
```

- Launch a bash process in the detached nginx instance

```
docker exec -it <id> /bin/bash
```

hello-world

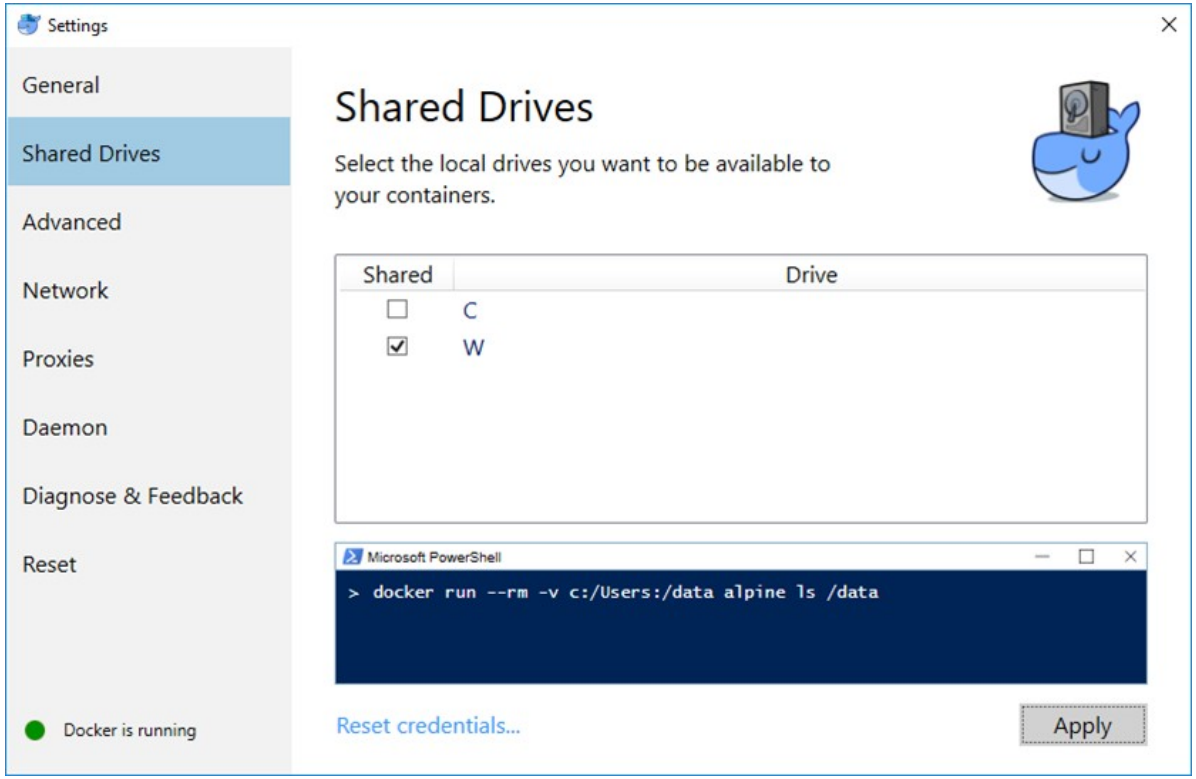
- Attach to the detached nginx instance

```
docker attach <id>
```

- Housekeeping commands

```
docker stop
docker rm
docker images
docker rmi
docker container prune
docker image prune
```

Exercise 2 – Externalities



Externalities

- Mounting file system volumes

```
docker run –it –v W:\data:/data debian:9 /bin/bash
```

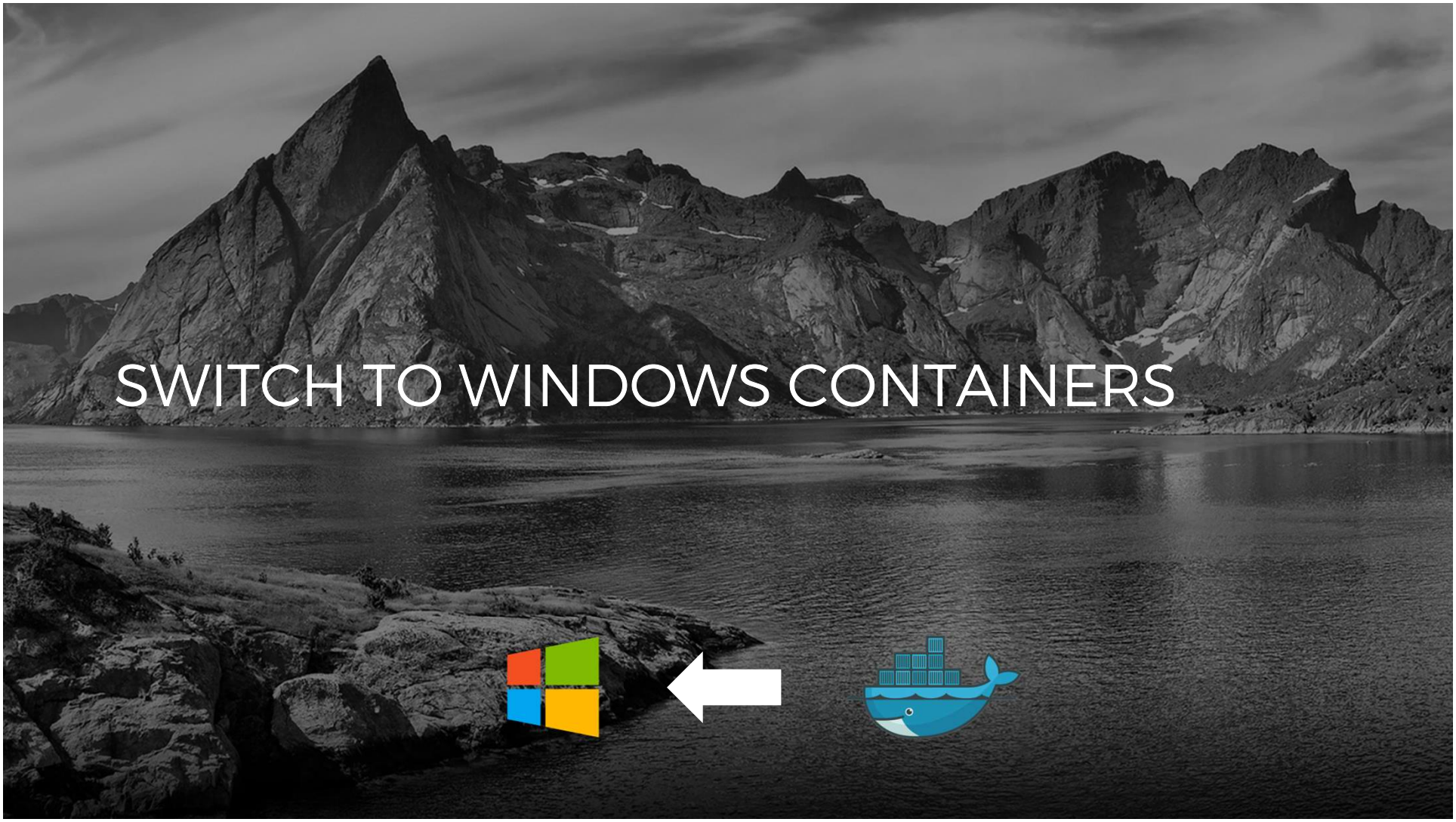
- Exposing ports

```
docker run -it -p 8080:80 nginx
```

- Environment variables

```
docker run -it -e "FOO=bar" debian:9 /bin/bash
root@8e035b9c48d9:/# echo $FOO
```


SERVE STATIC CONTENT IN IIS
(NANOSERVER)


SWITCH TO WINDOWS CONTAINERS

Serve static content in IIS (nanoserver)

- Review the base IIS image at https://hub.docker.com/r/microsoft/iis/
  - Note: nanoserver vs windowsservercore

- Start with the tutorial Dockerfile:

```
FROM microsoft/iis:nanoserver-10.0.14393.1715
RUN mkdir C:\site
RUN powershell -NoProfile -Command \
    Import-module IISAdministration; \
    New-IISSite -Name "Site" -PhysicalPath C:\site -BindingInformation
"*:8000:"
EXPOSE 8000
ADD content/ /site
```

Serve static content in IIS (nanoserver)

- Build

```
docker build -t my/iis .
```

- Run

```
docker run --rm -it --name iis my/iis
```

- Connect browse to <IP>:8000 – get IP from inspect:

```
docker inspect iis
docker inspect -f "{{ .NetworkSettings.Networks.nat.IPAddress }}" iis
```

Serve static content in IIS (nanoserver)

- Create a volume for our static site:

```
docker volume create --name website
```

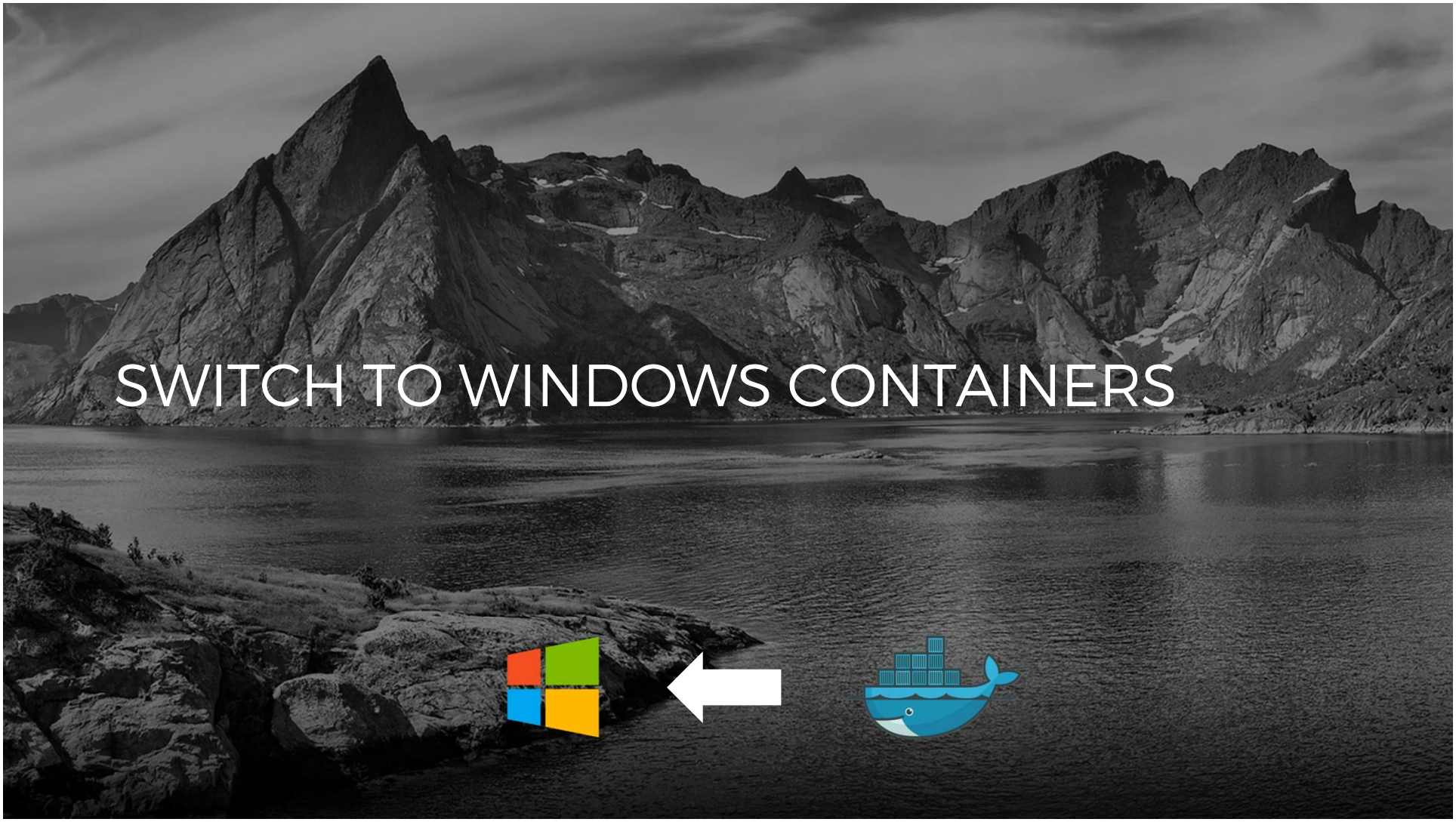
- Drop content into C:\ProgramData\Docker\volumes\website
- Remove ADD from Dockerfile and rebuild
- Run with external volume mounted

```
docker run --rm -it --name iis -v
C:\ProgramData\Docker\volumes\website:C:\site my/iis
```


DOCKERIZE A .NET CORE WEBAPI
MICROSERVICE (NANOSERVER &
LINUX)

Dockerize a .NET Core WebAPI microservice (nanoserver & linux)

Goal: Create a ASP.NET Core WebAPI microservice that runs identically under both nanoserver and linux


SWITCH TO WINDOWS CONTAINERS

Dockerize a .NET Core WebAPI microservice (nanoserver & linux)

- Review the tasksapp code
- Launch an interactive container for build

```
docker run --rm -it -v <your root path>\tasksapp\MyCo.Tasks:/build
microsoft/aspnetcore-build:2.0.0 /bin/bash
```

- Build

```
dotnet clean
dotnet restore
dotnet publish -c Release -o out
```

- Exit the “build” container

Dockerize a .NET Core WebAPI microservice (nanoserver & linux)

- Define the “runtime” container in a new Dockerfile

```
FROM microsoft/aspnetcore:2.0.0

RUN mkdir /service
WORKDIR /service

COPY MyCo.Tasks/out .

EXPOSE 80

ENTRYPOINT ["dotnet", "MyCo.Tasks.dll"]
```

- Build

```
docker build –t myco/tasks .
```

Dockerize a .NET Core WebAPI microservice (nanoserver & linux)

- Run

```
docker run --rm -it --name tasks -p:9871:80 myco/tasks
```

- Find out the container IP – due to the Windows Container networking flaw

```
docker inspect tasks
```

- Browse to http://<ip address>:80
- Kill the container

Better Builds with Docker

- Dockerfile.build approach

```
FROM microsoft/aspnetcore-build:2.0.0

WORKDIR /build

RUN dotnet clean
RUN dotnet restore
RUN dotnet publish -c Release -o out
```

- Build the build container image – this actually runs the build

```
docker build -t myco/tasks-build -f Dockerfile.build .
```

- Retrieve the build result from the image

```
docker create --name tasks-build myco/tasks-build
docker cp tasks-build:/build/out .
docker rm tasks-build
```

Better Builds with Docker

- From Docker 17.05 – multi-stage build approach

```
# Build Stage
FROM microsoft/aspnetcore-build:2.0-jessie AS tasks-build

RUN mkdir /build
WORKDIR /build
COPY MyCo.Tasks/ ./

RUN dotnet clean
RUN dotnet restore
RUN dotnet publish -c Release -o out

# Run Stage
FROM microsoft/aspnetcore:2.0-jessie

RUN mkdir /service
WORKDIR /service

COPY --from=tasks-build /build/out .

EXPOSE 80

ENTRYPOINT ["dotnet", "MyCo.Tasks.dll"]
```

- References
  - https://docs.docker.com/engine/userguide/eng-image/multistage-build/
  - https://blog.alexellis.io/mutli-stage-docker-builds/

Dockerize a .NET Core WebAPI microservice (nanoserver & linux)

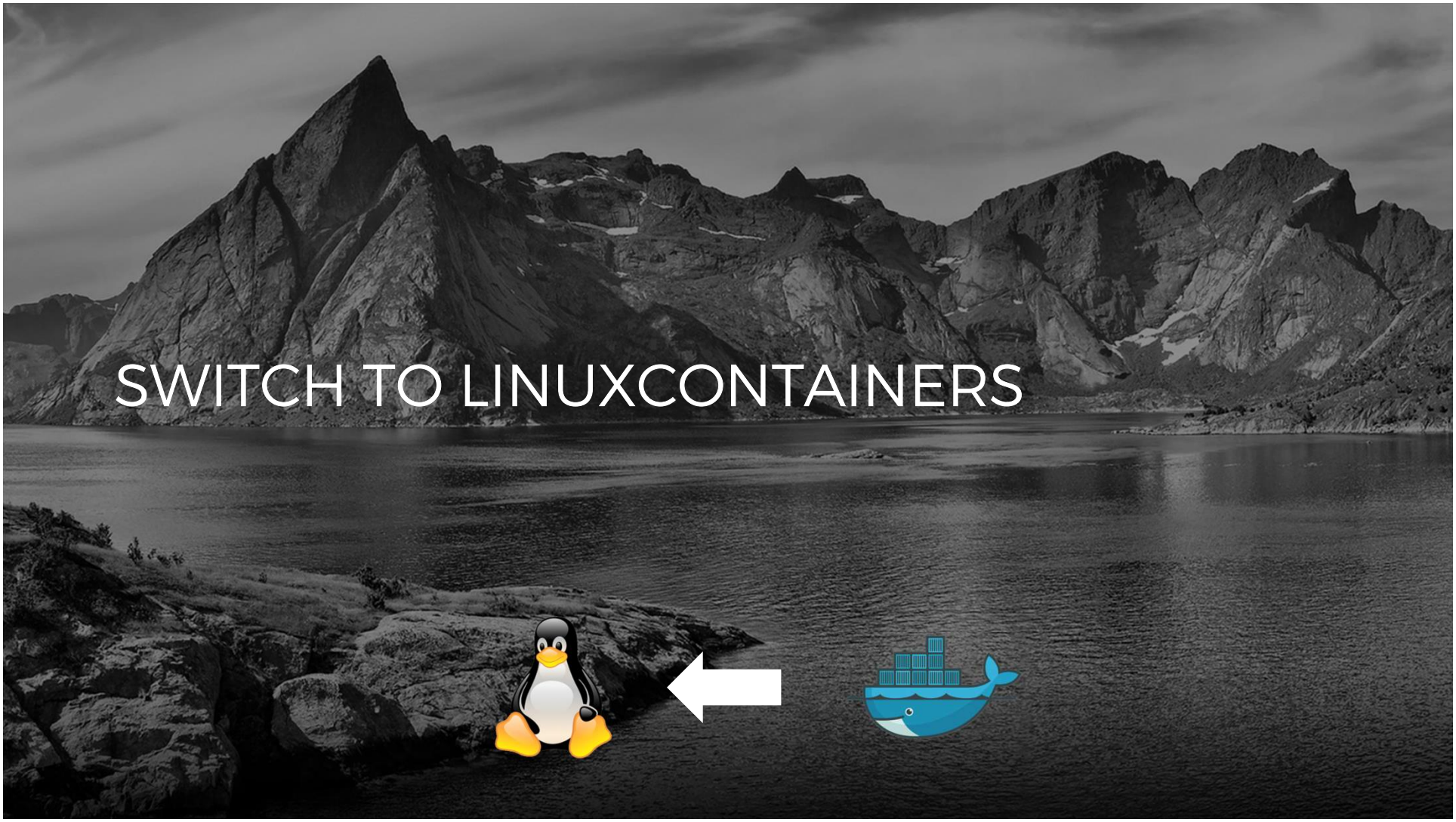
- Build and run

```
docker build –t myco/tasks .
docker run --rm -it --name tasks –p:9871:80 myco/tasks
```

- Find out the container IP – due to the Windows Container networking flaw

```
docker inspect tasks
```

- Browse to http://<ip address>:80
- Kill the container


SWITCH TO LINUXCONTAINERS

Dockerize a .NET Core WebAPI microservice (nanoserver & linux)

- Build and run

```
docker build –t myco/tasks .
docker run --rm -it --name tasks –p:9871:80 myco/tasks
```

- Browse to http://localhost:9871
- Kill the container

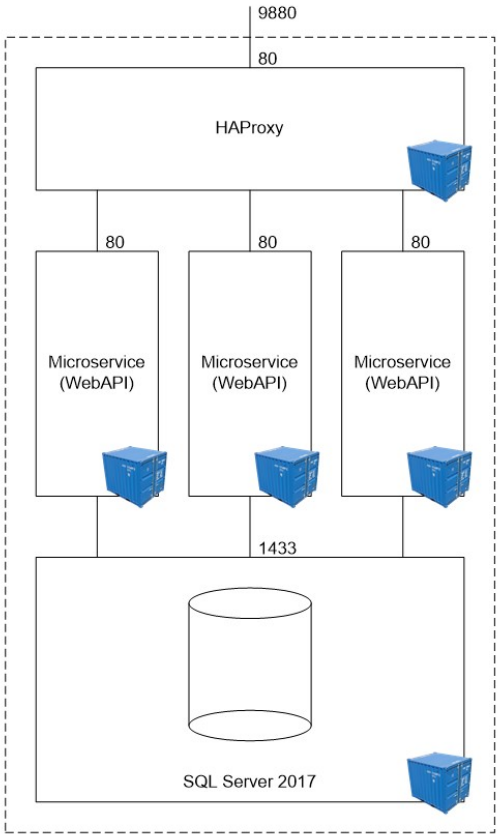

MULTI-CONTAINER SOLUTION
WITH DOCKER-COMPOSE (LINUX)

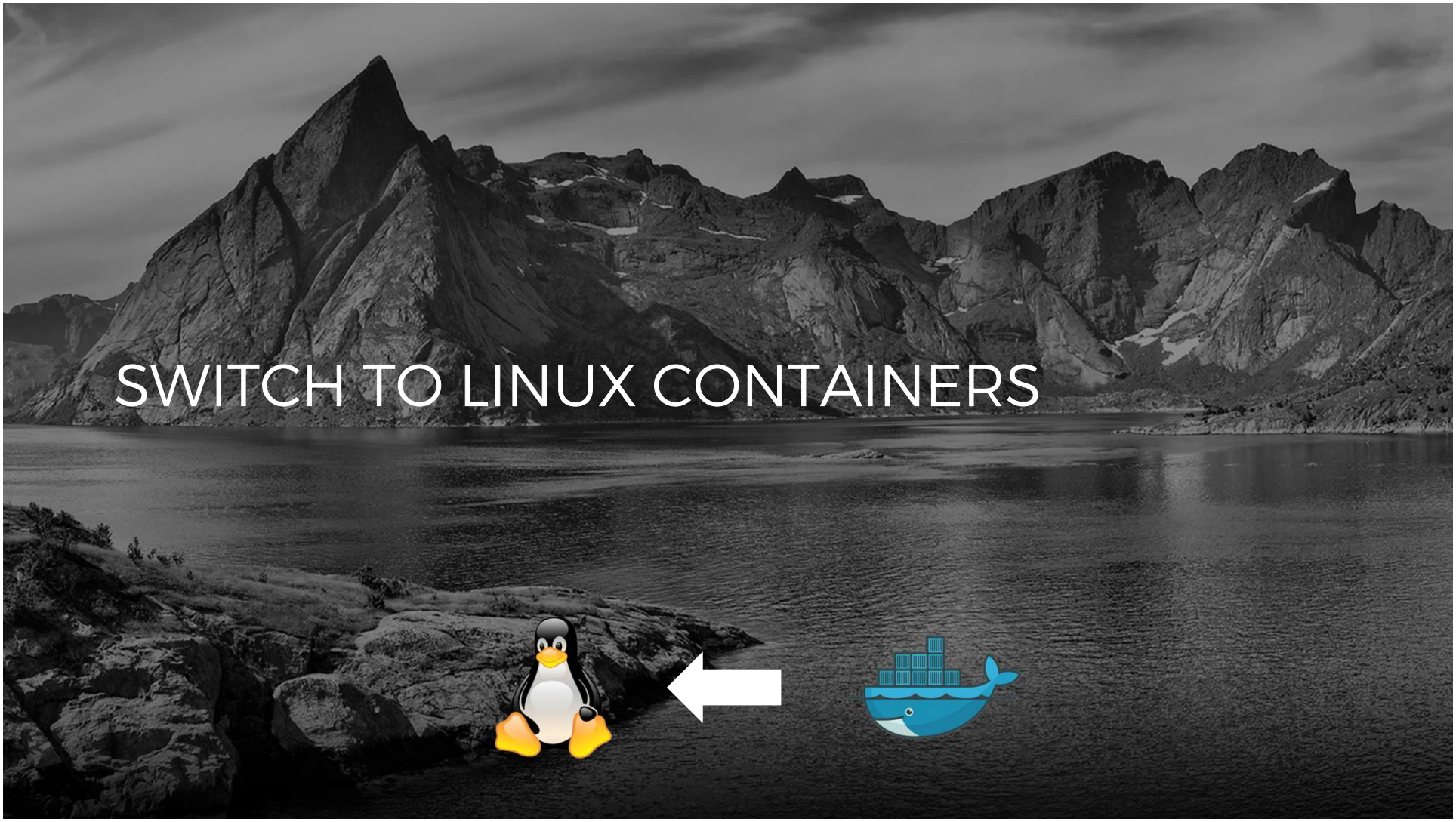
Multi-container solution with docker-compose (linux)

- docker-compose
  - Create multi-container stacks
  - Define stack in a single YAML file
  - Single command to launch all containers in the stack

```
docker-compose –f mystack.yml up
```

Multi-container solution with docker-compose (linux)




SWITCH TO LINUX CONTAINERS

Multi-container solution with docker-compose (linux)

- Initial docker-compose.yml

```
version: "3"
services:

    api:
        build: .
```

- Run and examine

```
docker-compose up
```

- Destroy

```
docker-compose down
```

Multi-container solution with docker-compose (linux)

- Map port

```
ports:
    - "9871:80"
```

- Run again and browse http://localhost:9870/api/tasks

Worker Nodes

- Build out three API worker nodes

```
version: "3"
services:

    api1:
        build: .
        ports:
            - "9871:80"

    api2:
        build: .
        ports:
            - "9872:80"

    api3:
        build: .
        ports:
            - "9873:80"
```

Worker Nodes

- Explicitly label the images built by docker-compose:

```
version: "3"
services:

    api1:
        image: myco/tasks
        build: .
        ports:
            - "9871:80"

    api2:
        image: myco/tasks
        build: .
        ports:
            - "9872:80"

    api3:
        image: myco/tasks
        build: .
        ports:
            - "9873:80"
```

Worker Nodes

- Browse to
  - http://localhost:9871/api/tasks
  - http://localhost:9872/api/tasks
  - http://localhost:9873/api/tasks

- Stop the stack

```
docker-compose –f docker-compose.lin.yml down
```

Load Balancer

- Add haproxy service

```
haproxy:
        image: library/haproxy:1.7
        volumes:
          - haproxy_cfg:/usr/local/etc/haproxy
        ports:
            - "9880:80"
            - "9881:70"
        links:
            - api1
            - api2
            - api3
```

Load Balancer

- Declare the volume

```
volumes:
  haproxy_cfg:
    external: true
```

- Launch – observe volume error

```
docker-compose up

ERROR: Volume haproxy_cfg declared as external, but could not be found.
Please create the volume manually using `docker volume create --
name=haproxy_cfg` and try again.
```

Load Balancer

- Create the volume to hold the haproxy configuration:

```
docker volume create haproxy_cfg
```

- Import the haproxy.cfg file into the volume via a temporary container:

```
scripts\haproxy_copy_config.cmd
```

Load Balancer

- Run and observe roundrobin load balancing

```
docker-compose up
```

- Browse to http://localhost:9880/api/tasks
- Each node has its own in-mem data, so each refresh will be different
- Drop the worker-node ports

Load Balancer

- Create docker-compose.lin.dev.yml to mixin worker-node ports for debugging

```
version: "3"
services:

    api1:
        ports:
            - "9871:80"

    api2:
        ports:
            - "9872:80"

    api3:
        ports:
            - "9873:80«
```

Launch stack with ports mixed in

```
docker-compose -f docker-compose.lin.yml -f docker-compose.lin.dev.yml up
```

Add SQL Server 2017

- Create a data directory – e.g mine is W:\data
- Launch a SQL Server 2017 instance – in Docker!

```
docker run ^
     -e "ACCEPT_EULA=Y" ^
     -e "MSSQL_SA_PASSWORD=p@ssw0rz!@#" ^
     -p 1401:1433 ^
     -v W:\data:/var/opt/mssql ^
     --name sql1 ^
     -d ^
     microsoft/mssql-server-linux:2017-GA
```

- Shortcut scripts/start_sql_linux.cmd

Add SQL Server 2017

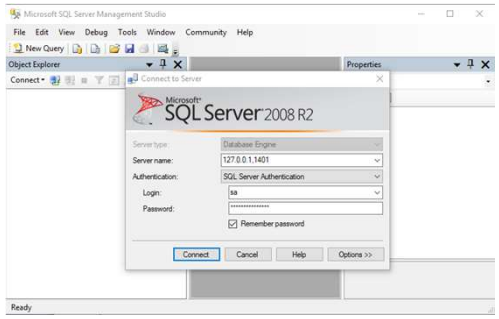
- Edit the microservice config to connect to a database
- Create the database on a temporary SQL Server instance
  - Create a volume for the data files

```
docker volume create tasks-db
```

  - Review and run the script:

```
scripts\start_sql_linux.cmd
```

  - Connect via SSMS



Add SQL Server 2017

- Create a new database with the script at Database\tasks_database.sql
- Terminate SQL Server

```
docker stop sql1
```

Add SQL Server 2017

- Add a new “db” service to docker-compose.lin.yml:

```
db:
  image: microsoft/mssql-server-linux:2017-GA
  environment:
    ACCEPT_EULA: "Y"
    MSSQL_SA_PASSWORD: "p@ssw0rz!@#"
  volumes:
    - tasks-db:/var/opt/mssql
  expose:
    - "1433"
  ports:
    - "1402:1433"
```

Add SQL Server 2017

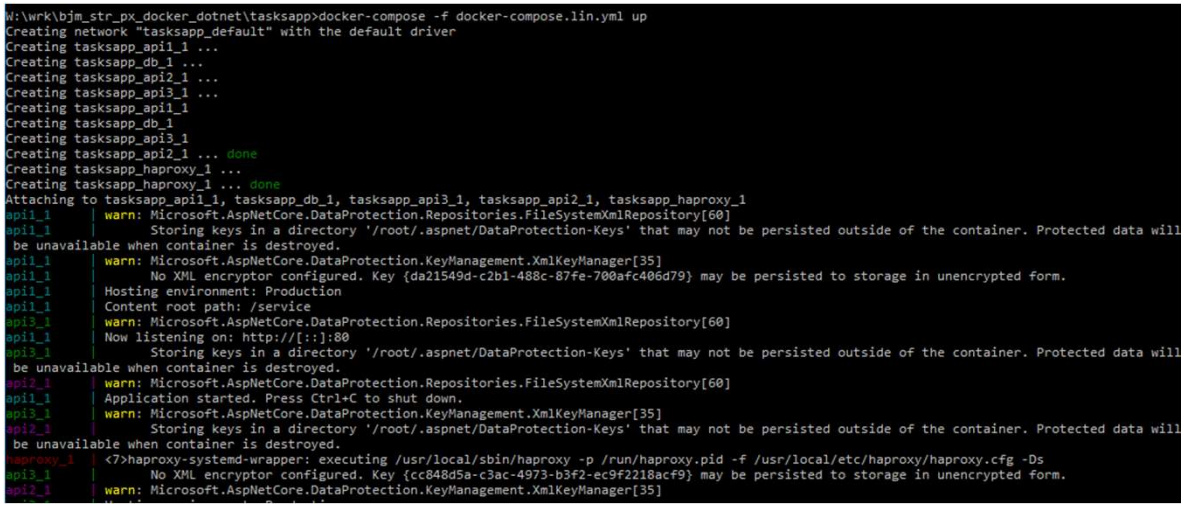
- Declare the external volume

```
volumes:
    haproxy_cfg:
        external: true
    tasks-db:
        external: true
```

Add SQL Server 2017

- Launch the stack

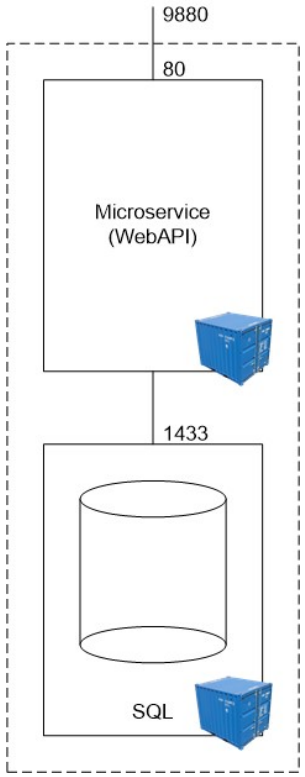
```
docker-compose -f docker-compose.lin.yml up
```

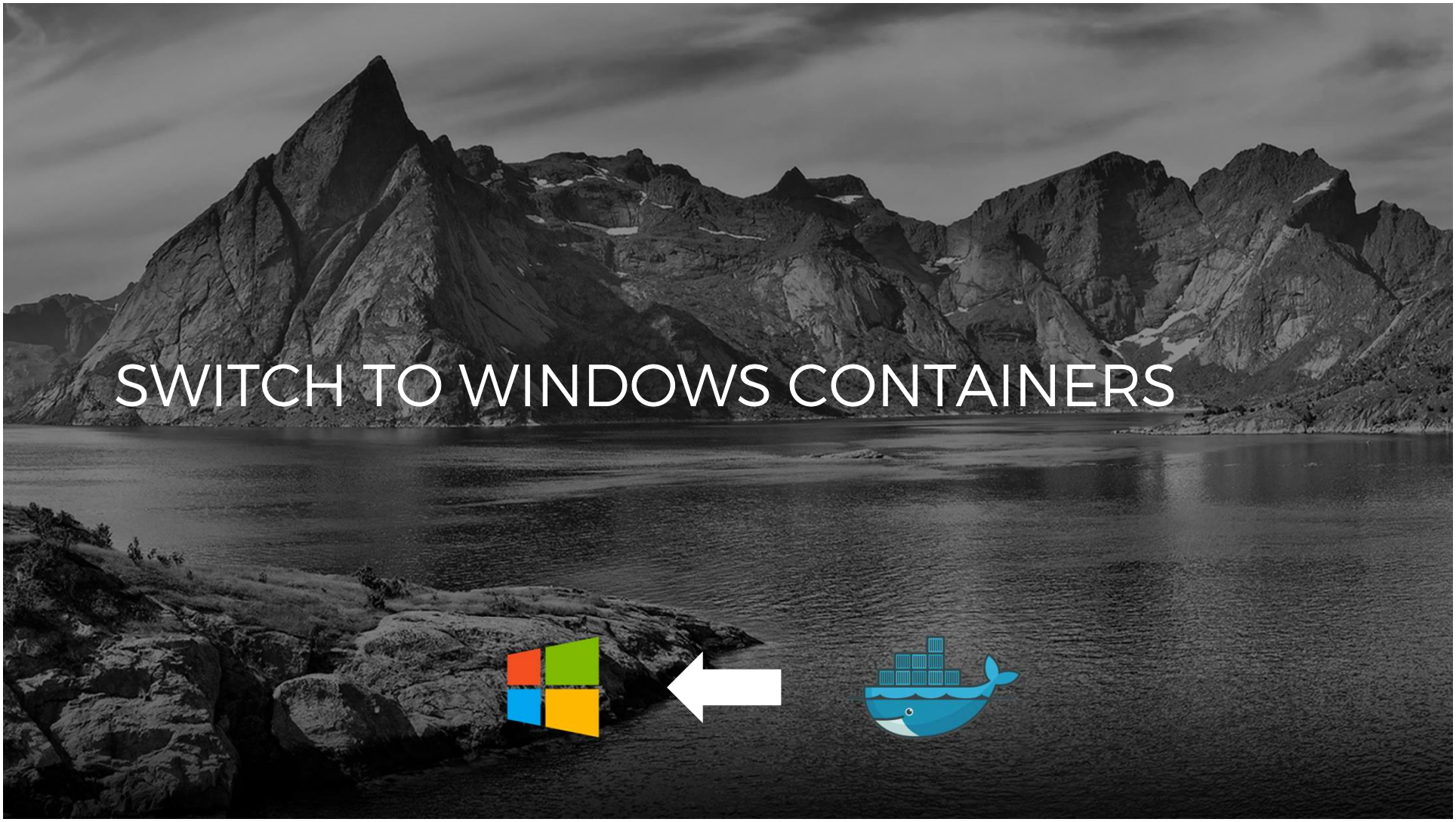


Browse to http://localhost:9880/api/tasks


MULTI-CONTAINER SOLUTION
WITH DOCKER-COMPOSE
(NANOSERVER)

Multi-container solution with docker-compose (nanoserver)




SWITCH TO WINDOWS CONTAINERS

Create Database

- Create the managed volume

```
docker volume create tasks-db
```

- Launch SQL Server 2017 on Windows via helper script

```
scripts\start_sql_win.cmd
```

- Connect – note NAT limitation for Windows Containers
- Find IP of the container on the host-only-network

```
docker inspect sql1
```

- References
  - https://blog.sixeyed.com/published-ports-on-windows-containers-dont-do-loopback/
  - https://blogs.technet.microsoft.com/virtualization/2016/05/25/windows-nat-winnat-capabilities-and-limitations/

Define Single Worker Node

```
version: "3"
services:

    api:
        build: .
        image: myco/tasks
        environment:
            ASPNETCORE_ENVIRONMENT: "Production«
        ports:
            - "9871:80"
```

Define Database

- Database service:

```
 db:
        image: microsoft/mssql-server-windows-developer:2017
        environment:
            ACCEPT_EULA: "Y"
            SA_PASSWORD: "p@ssw0rz!@#"
            ATTACH_DBS: "[{'dbName':'Tasks','dbFiles':['C:\\\\data\\\\Tasks.mdf',
'C:\\\\data\\\\Tasks.ldf']}]"
        volumes:
            - "tasks-db:C:\\data"
        ports:
            - "1401:1433"
```

- Declare volume

```
volumes:
    tasks-db:
        external: true
```

Launch Stack

```
docker-compose -f docker-compose.win.yml up
```

```
W:\wrk\bjm_str_px_docker_dotnet\tasksapp>docker-compose -f docker-compose.win.yml up
Creating tasksapp_api_1 ...
Creating tasksapp_db_1 ...
Creating tasksapp_api_1
Creating tasksapp_db_1 ... done
Attaching to tasksapp_api_1, tasksapp_db_1
api_1  | Hosting environment: Production
api_1  | Content root path: C:\service
api_1  | Now listening on: http://[::]:80
api_1  | Application started. Press Ctrl+C to shut down.
db_1   | VERBOSE: Starting SQL Server
db_1   | VERBOSE: Changing SA login credentials
db_1   | VERBOSE: Attaching 1 database(s)
db_1   | VERBOSE: Invoke-Sqlcmd -Query IF EXISTS (SELECT 1 FROM SYS.DATABASES WHERE NAME
db_1   |  = 'Tasks') BEGIN EXEC sp_detach_db [Tasks] END;CREATE DATABASE [Tasks] ON
db_1   | (FILENAME = N'C:\data\Tasks.mdf'),(FILENAME = N'C:\data\Tasks.ldf') FOR ATTACH;
db_1   | VERBOSE: Started SQL Server.
db_1   |
```


SCHEDULING A CLUSTER WITH
DOCKER SWARM

Scheduling a cluster with Docker Swarm

- Docker Swarm is a Docker-native clustering system that exposes the same API as the standalone Docker Engine
- References:
  - https://docs.docker.com/compose/swarm/


DEPLOYING TO AWS ECS

Deploying to AWS ECS

- TODO


TOOLING ETC

CI and Build Containers

- TODO

Testing Containers

- TODO

Docker Cloud

- TODO

Visual Studio 2017 Support for Docker

- TODO


b@bren.cc
http://u.bren.cc/github
brendon.matheson
http://u.bren.cc/linkedin
http://u.bren.cc/youtube
http://u.bren.cc/twitter