Kenny R N Wilson

Docker

Introduction

THIS DOCUMENT COVERS

- Introduction

Why Docker?

Different services can require different versions of the OS and other dependencies such as the runtimes, programming languages and libraries. It is unlikely that any single service will require the exact set of dependencies installed on the host machine. Docker is a containerization technology that makes it possible to run multiple services on the same host machine, each with the isolated set of dependencies it needs to run. Docker makes it possible to run the same service in the same environment no matter where and when we run it. The Docker containers are shielded from upgrades and downgrades on the host machine.

Cheat Sheets

Docker commands

Run

	Column Header
Start a container	`docker run <img-name>`
Directory Mapping*	`docker run -v <host-dir> <cont-dir> <img-name>`
Port Mapping	`docker run -p <host-port>:<container-port> <image-name>`
Name the container	`docker run  <img-name> --name <cont-name>`

The directory mapping can be a little tricky to get correct, especially when the host operating system is windows, and the container is Linux. Let us look at some examples.

Path – Absolute Windows

The below command maps the windows folder `C:\Users\kenne\outputs` to the container folder `/tmp/logs`.

```
docker run -d -v C:\Users\kenne\outputs:/tmp/logs -e
"Serilog__WriteTo__0__Args__path=/tmp/logs/log.txt" -p 5050:80 dirmapping
```

Path – Relative Windows

The below command maps the folder logfiles under the host folder the container was run from to the container folder `/tmp/logs`.

```
docker run -d -v "%CD%/logfiles:/tmp/logs" -e
"Serilog__WriteTo__0__Args__path=/tmp/logs/log.txt" -p 5050:80 dirmapping
```

Copy

The following shows how to copy a file from a docker container to the host. 306 is the first few characters of the docker container id.

```
  docker cp 306:/var/opt/mssql/data/JsonObjects.bak C:\Users\rps\
```

If we want to copy a file from the host to the container, we simply reverse the process.

```
  docker cp C:\Users\rps\JsonObjects.bak 306:/var/opt/mssql/data/
```

Docker Inspect

We can list the entire information for a container using. This also gives the container IP Address.

```
docker inspect <container-id>
```

Docker Exec

If the container is Linux based, we can execute a shell to log on and see what is happening on a running container.

```
  docker exec -i -t 7d bash
```

We now have a shell onto the container.

Docker Compose

Run

	Column Header

Map Windows Relative Dir	`volumes:`
	` - ./keycloakserver/kcdata/:/tmp/kcdata/`

Port Mapping	`ports:`
	`  - 8050:80`
	`  - 8051:443`

Environment Variables	`environment:`
	` KEYCLOAK_USER : admin`
	` KEYCLOAK_PASSWORD : admin`

DotNet Core Env Variables	`environment:`
	` KeyCloak__Realm : http://kc:8080/auth/realms/kennyrealm`

Docker Theory

Overview of Docker

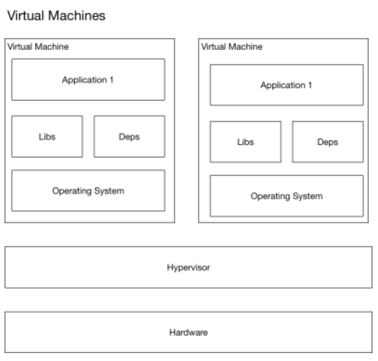
Host

In Docker terminology the machine we are running Docker on is known as the host. Docker runs on the host OS and manages multiple containers.

Container

Docker is a container technology. A container is a standardised package of software and all the dependencies needed to run that software. These dependencies include specific versions of programming languages, runtimes, and libraries. The benefit of using a container is that we get the exact same environment every time we run the container. The behaviour does not change when we upgrade or downgrade dependencies on the host machine.

All docker containers must use the same OS kernel. This kernel, however, does not have to be the same as the host operating system. Virtualization and WSL 2.0 technologies enable us to run Linux containers on a Windows machine.



Containers Versus Virtual Machine

Containerization and virtual machines are complementary rather than competing technologies. Virtual machines provide a heavyweight solution requiring large amounts of disk space and long startup times. The benefit of VMs is that they provide complete isolation and each VM can run a different OS Kernel.

Containers are a lightweight solution with each container requiring only Megabytes of disk space. They start quickly; however they have less isolation and all containers on the host must use the same OS Kernel (although they can have different Operating Systems that share the same kernel)

Images

A docker image is a template that is used to create docker containers. Each container is hence an instance of the image. A Docker image is created using the docker build command. We now show how to build a simple image that can be used to run Node.js webservers inside containers.

Docker Tutorial

Containers and Images

We use Docker to provide the instructions that docker uses to build an image. To illustrate how we go about creating images and containers, consider the following example which creates a simple application that runs a node.js express web server. The full source code is available here.

Code

BUILD AND RUN STANDALONE

Before we can Dockerize a service we must first know how to build and run the application outside of docker. First, we create a single source code file to contain the webserver code.

Listing 1 index.js

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {res.send('Hello Kenny W!')})
app.listen(port);
```

We need a package.json file to specify the single dependency.

Listing 2 package. json

```
{
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

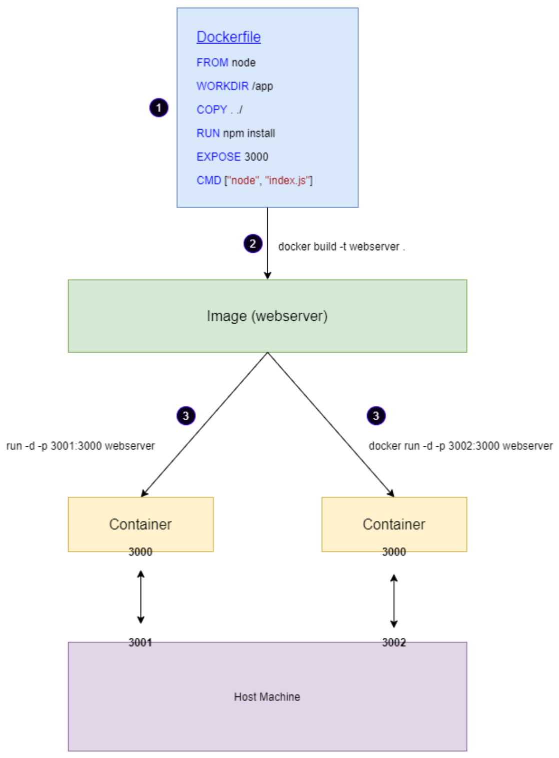
We update the dependencies and run the application very simply as follows.

```
npm install
node index.js
```

We can then open a web browser and entering the URI `http://localhost:3000/`

Dockerize

The following diagram highlights the way in which we create a docker image and then use that image to start two containers that listen on different ports.



1. Create a Dockerfile.

We need to write a Docker file that describes how to build a docker image.

```
FROM node
WORKDIR /app
COPY . ./
RUN npm install
EXPOSE 3000
CMD ["node", "index.js"]
```

2. Build an image from the Dockerfile.

We build the image and give it a name using the following command. The -t argument enables us to give the image a name.

```
docker build -t webserver .
```

The `images` command can then be used to check our built image.

```
docker images

> REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
> webserver     latest    c6eeaee89aa5   7 seconds ago    941MB
```

3. Use the image to start two containers.

We now use our image to create two containers that listen on different ports. There are two important points to note in the commands below.

1. -d option to run detached like a background process.
2. -p to map ports on the container to ports on the host.

```
docker run -d -p 3001:3000 --name server_one webserver
docker run -d -p 3002:3000 --name server_two webserver
```

The `ps` command can then be used to check our two running containers (I have removed some columns of the display for clarity)

```
  > docker ps

  CONTAINER ID   IMAGE       PORTS                       NAMES
  8f0f3c7f5e79   webserver   0.0.0.0:3002->3000/tcp      server_two
  c6eab858fb5a   webserver   0.0.0.0:3001->3000/tcp      server_one
```

Dockerize ASP.Net core

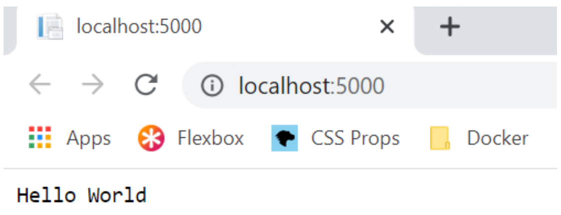
Let us build on the previous section by using Docker to containerize a .NET Core Webserver. The full source code is in SourceCode

Without Docker

```
public class Program
{
    public static void Main(string[] args)
    {
        string message = args.Length > 0 ? args[0] : "Hello";
        var webHost = new WebHostBuilder()
            .UseKestrel()
            .Configure(app =>
            {
                app.Run(async ctx =>
                {
                    await ctx.Response.WriteAsync(message);
                });
            })
            .Build();

        webHost.Run();
    }
}
```

Running this application inside of Visual Studio and connecting to http://localhost:5000 gives the following.



```
Hello World
```

With Docker

Let us now work through step by step the process of creating a docker image that will run our ASP.NET Core application inside a container. We will use some tricks that will enable us to log onto the container and see what has happened at each stage. All the instructions we use are documented here.

https://docs.docker.com/engine/reference/builder/#workdir

The complete Dockerfile is given in the solution as follows.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
RUN dotnet restore
COPY ASPHelloWorld/. ./ASPHelloWorld/
WORKDIR /kennyswebapp/ASPHelloWorld
RUN dotnet publish -c Release -o out
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS runtime
WORKDIR /mydeploydir
COPY --from=build /kennyswebapp/ASPHelloWorld/out ./
ENTRYPOINT ["dotnet", "ASPHelloWorld.dll"]
```

We build the image as follows.

```
docker build -t dotnetserver .
```

And check the built image as follows.

```
docker images

> REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
> dotnetserver   latest    44dce4484a4c   2 minutes ago    205MB
```

Now we run the image as follows.

```
docker run -d -p 8080:80 --name myapp dotnetserver
```

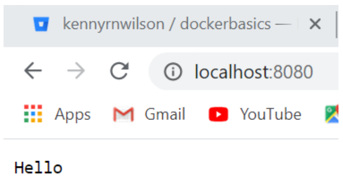
Note three important points.

- We specify the image name `dotnetserver`
- We specify the container name `--name myapp`
- We map the port on the container to the port on the host `-p 8080:80`
- We run the server detached `-d`

```
docker ps -a

CONTAINER ID   IMAGE          COMMAND                   PORTS                  NAMES
29f478c4db1b   dotnetserver   "dotnet ASPHelloWorl…"    0.0.0.0:8080->80/tcp   myapp
```

We can then see the application on port 80 on the host.



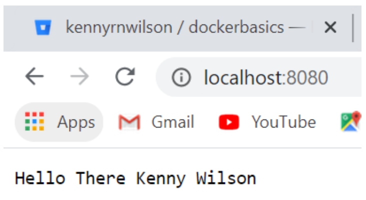
```
Hello
```

Overriding The Applications Arguments In Docker

We can pass in arguments to our application by passing in arguments to the command line.

```
docker run -d -p 8080:80 --name myapp dotnetserver  "Hello There Kenny Wilson"
```

And checking the server we see



```
Hello There Kenny Wilson
```

Dockerfile In Detail

Let us take a more detailed look at the Dockerfile now.

FROM

All Dockerfile files must start with a `FROM` instruction. The `FROM` instruction initializes a new build stage and specifies the base image for the instructions that follow it.

> **Base Image**
>
> An image with no parent image. Its first line will be `FROM scratch`

If we give the `FROM` instruction an optional `AS` argument, we can assign a name to this build stage which enables us to refer to it from subsequent `FROM` and `COPY` instructions. Let us go ahead and set the first line of our Dockerfile. We will specify the .NET sdk as the base image for the first stage and name it `build`.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
```

WORKDIR

The `WORKDIR` instruction sets the working directory for subsequent `RUN, CMD, ENTRYPOINT, COPY and ADD` instructions. If it does not exist it will be created. We can call the command multiple times in which case subsequent relative paths are relative to the path of the previous `WORKDIR` command.

Let us go ahead and set the working directory that we want to work in on the docker container.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
```

After this stage and each subsequent stage, we will create an image from the partial Dockerfile up to that point and run it up so we can log onto the container and take a look at what is does.

```
docker build -t  aspdotnetapp .
docker run -d aspdotnetapp sleep 120
docker exec -i -t d bash


>> root@d5c7d231d835:/kennyswebapp#
```

COPY

The `COPY` instruction copies files and directories from the host to folders on the container. We now add logic to our Dockerfile to copy over the solution file to the container.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
```

If we do out little test exercise, we expect to see a .sln file on the container in the /kennysapp folder.

```
docker build -t  aspdotnetapp .
docker run -d aspdotnetapp sleep 120
docker exec -i -t be7  bash

>> root@be7bd4d1026c:/kennyswebapp# ls
>> ASPHelloWorld  ASPHelloWorld.sln
```

RUN

Execute a given command into a new layer on top the current image and commit the results. We use the run command to get NuGet to restore dependencies.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
RUN dotnet restore
```

We now use our little test exercise to see the results. Notice the NuGet results in the `APSHelloWorld/obj/` folder.

```
>> root@ff8594c555af:/kennyswebapp# ls ASPHelloWorld/obj/
>> ASPHelloWorld.csproj.nuget.dgspec.json  project.assets.json
>> ASPHelloWorld.csproj.nuget.g.props      project.nuget.cache
>> ASPHelloWorld.csproj.nuget.g.targets
```

COPY (The Source Code)

We now add another copy to copy over the source code from the host to the container.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
RUN dotnet restore
COPY ASPHelloWorld/. ./ASPHelloWorld/
```

Build and publish the app

We now use add lines to build the application and publish the application.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
RUN dotnet restore
COPY ASPHelloWorld/. ./ASPHelloWorld/
WORKDIR /app/ASPHelloWorld
RUN dotnet publish -c Release -o out
```

FROM

FROM creates a new stage and sets a new base image. This time we want the base image to be the .NET runtime rather than the .NET SDK we were using in the previous stage.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
RUN dotnet restore
COPY ASPHelloWorld/. ./ASPHelloWorld/
WORKDIR / kennyswebapp /ASPHelloWorld
RUN dotnet publish -c Release -o out
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS runtime
WORKDIR /app
```

COPY

We need to access the directories from the previous stage from the new stage, so we use a special form of copy.

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
RUN dotnet restore
COPY ASPHelloWorld/. ./ASPHelloWorld/
WORKDIR /kennyswebapp/ASPHelloWorld
RUN dotnet publish -c Release -o out
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS runtime
WORKDIR /mydeploydir
COPY --from=build /kennyswebapp/ASPHelloWorld/out ./
```

Notice how we reference the name we assigned in line 1 using the AS argument. If we run our little test exercise, we should now have a deployed application.

```
>> root@dc5e8b374be2:/mydeploydir# ls
>> ASPHelloWorld              ASPHelloWorld.pdb
>> appsettings.json
>> ASPHelloWorld.deps.json    ASPHelloWorld.runtimeconfig.json  web.config
>> ASPHelloWorld.dll          appsettings.Development.json
```

Start up our application.

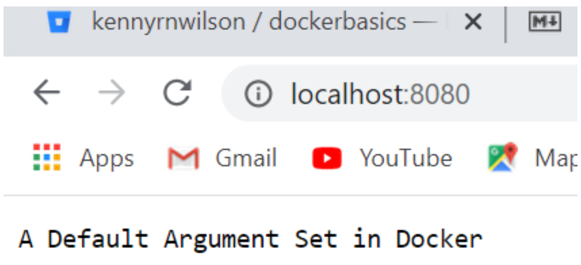
The final step starts our application

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /kennyswebapp
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
RUN dotnet restore
COPY ASPHelloWorld/. ./ASPHelloWorld/
WORKDIR /kennyswebapp/ASPHelloWorld
RUN dotnet publish -c Release -o out
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS runtime
WORKDIR /mydeploydir
COPY --from=build /kennyswebapp/ASPHelloWorld/out ./
ENTRYPOINT ["dotnet", "ASPHelloWorld.dll"]
CMD ["A Default Argument Set in Docker"]
```

If we rebuild the image, we are ready to run it. We need one more thing. the docker image runs in the container on port 80. The launchSettings.json is ignored. For this reason, we map port 80 on the container to port 8080 on the container host so we can access it from localhost.

```
docker run -d -p 8080:80 --name myapp aspnetapp
```

When we connect to our application from a browser, we see the value of the CMD is returned.



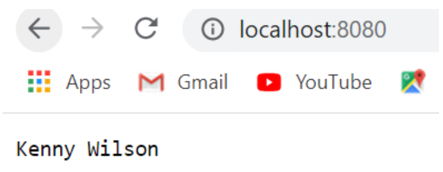
```
A Default Argument Set in Docker
```

https://stackoverflow.com/questions/48669548/why-does-aspnet-core-start-on-port-80-from-within-docker

Overriding value of CMD

If we want to override the value of CMD we can do this from the command line. We create a second instance on a different port and with a different message.

```
docker run -d -p 8081:80 --name myapp2 aspnetapp "Hello Kenny"
```



```
Kenny Wilson
```

Mapping directory from docker container to host.

Now consider the situation where our application logs to the container via Serilog. The full source code is at SourceCode. The main points are that we configure the application to log to a file on the container.

Serilog (appsettings.config)

```
"WriteTo": [
  {
    "Name": "File",
    "Args": {
      "path": "/var/tmp/logs/log.txt",
      "rollingInterval": "Day"
    }
  },
  {
    "Name": "Console"
  }
]
```

We create the directory `/var/tmp` in the Dockerfile.

Dockerfile

```
WORKDIR /app
COPY *.sln .
COPY ASPHelloWorld/*.csproj ./ASPHelloWorld/
RUN dotnet restore
COPY ASPHelloWorld/. ./ASPHelloWorld/

WORKDIR /app/ASPHelloWorld
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS runtime
WORKDIR /app
RUN mkdir -p /var/tmp/logs
COPY --from=build /app/ASPHelloWorld/out ./

ENTRYPOINT ["dotnet", "ASPHelloWorld.dll"]
```

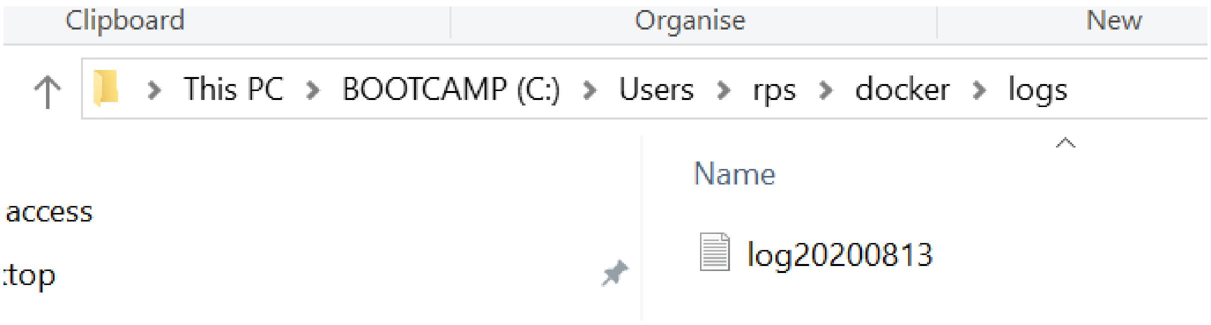
We now build the container.

```
docker build -t aspdotnetapp .
```

Finally, we run the container and map the container log file `/var/tmp/logs` to the folder `C:\Users\rps\docker\logs` on the container host

```
docker run -d -p 8080:80 -v C:\Users\kenne\docker\logs:/var/tmp/logs
aspdotnetapp
```

When we attach a browser to the port localhost:8080 we will see the log file shows up in the docker host filesystem in directory `C:\Users\rps\docker\logs`



log20200813

We can also view the mount that maps the directory by running the docker inspect command on the running container

```
Docker inspect 7d
```

And inside the output we see

```
        "Mounts": [
            {
                "Type": "bind",
                "Source": "/host_mnt/c/Users/rps/docker/logs",
                "Destination": "/var/tmp/logs",
                "Mode": "",
                "RW": true,
                "Propagation": "rprivate"
            }
        ],
```

A docker image captures the private filesystem that the that your containerized component will run in. The image contains just what the component needs. The image isolates all the dependencies your component needs.

A Dockerfile describes how to assemble a private filesystem for a container. This file provides step by step instructions on how to build up the image.

ASP.NET Core, Docker, and HTTPS

We need to take care of a few different things when running an ASP.NET Core application from docker using HTTPS. The following article from Microsoft describes the basics Docker-https

I have created a snippet that shows this working with docker-compose.

Docker Compose, Environment Variables and ASP.NET Core

Consider the following `docker-compose.yaml` file.

```
version: '3'
services:
 webserver:
  build:
   context: .
  environment:
   ASPNETCORE_Kestrel__Certificates__Default__Password : secret
   ASPNETCORE_Kestrel__Certificates__Default__Path : /https/aspnetapp.pfx
   ASPNETCORE_URLS : https://+;http://+
   ASPNETCORE_HTTPS_PORT : 8002
   Logging__LogLevel__Microsoft : Error
  volumes:
   - "${USERPROFILE}\\.aspnet\\https:/https"
  ports:
    - 8000:80
    - 8001:443
```

The first thing to note is that paths in ASP.NET core are separated by : but this is not suitable for environment variables so we use the __ separator instead. `"Logging:LogLevel:Microsoft"` becomes `Logging__LogLevel__Microsoft`. The second thing to note is a feature of ASP.NET Core and not Docker. We prefix anything that configures the host with **ASPNETCORE_.**

Finally, in docker the environment section can be a map or an array.

```
version: '3'
services:
 webserver:
  build:
   context: .
  environment:
   - ASPNETCORE_Kestrel__Certificates__Default__Password=secret
   - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx
   - ASPNETCORE_URLS=https://+;http://+
   - ASPNETCORE_HTTPS_PORT=8003
   - Logging__LogLevel__Microsoft=Error
  volumes:
   - "${USERPROFILE}\\.aspnet\\https:/https"
  ports:
    - 8000:80
    - 8001:443
```

Docker Commands

Docker Run

As mentioned earlier each container is an instance of an image. A container is created using the docker run command.

Tags

If you look up an image on Dockerhub.com you will find all the supported tags for that image. Tags enable us to specify different versions of the image.

```
docker run redis:4.0
```

StdIn/StdOut

By default, docker containers run in a non-interactive form. They do not listen to stdin. In this form there is no terminal to read input from. To provide input we must map stdin from our host to the docker container.

```
docker run -i <image>
```

If we want a prompt and terminal, we need to add a terminal too.

```
docker run -i -t <image>
```

Port Mapping

Consider we run a simple webapp.

```
docker run kodekloud/webapp
```

If we want to know the IP of a docker container we run the command.

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
$INSTANCE_ID
```

If we want to map a port from our docker container to the docker host, we run the command as follows

```
docker run -p80:5000  kodekloud/webapp
```

Running Commands

Containers are meant to run a specific task or process e.g. to host an instance of a web server or database. Once the task is complete the container exits. The container only lives if the process inside it is alive. Once a web server stops the container exits. We can instruct docker to run a command on our container.

```
docker run ubuntu  sleep 120
```

Now we can execute a command on this ubuntu container while it is running.

```
C:\Users\rps>docker exec ce55  cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      ce5539c1b552
```

Docker Compose

Linking Containers

Consider a simple example where one container runs an instance of MongoDB and other container runs a script that uploads data. The upload container needs to have access to the MongoDB container IP address. The way we do this is with links. Consider the following. The bold pieces link together the containers for mongohost.

docker-compose.yaml

```
version: '3'
services:
    mongohost:
        image: mongo

        ports:
            - 27010:27017
    mongo-load:
        build:
            context: MongoPopulate
            dockerfile: Dockerfile
        links:
            - mongohost
```

MongoPopulate/Dockerfile

```
FROM mongo
COPY myarchive /myarchive
COPY populate.sh /populate.sh
CMD "./populate.sh"
```

MongoPopulate/populate.sh

```
echo "Doing the load"
sleep 10
mongorestore --uri="mongodb://mongohost:27017" --drop --archive=myarchive
--gzip
echo "Finished"
```

Installation

Docker For Windows

I am installed Docker Community Edition for Windows.

https://hub.docker.com/editions/community/docker-ce-desktop-windows

There are two ways to use Docker on windows. Docker Toolbox is a legacy application, so we only consider Docker desktop for windows.

DOCKER DESKTOP FOR WINDOWS

Docker for windows uses the windows virtualization technology Hyper V. Because of this dependency Docker for Windows requires Windows Professional or Enterprise. The default option is to run Linux underneath. In this configuration all containers are Linux containers.

There is now also an option for Windows Container where each container runs on windows.

Introduction Questions

Why docker?

Multiple services require difference versions of the OS or different versions of dependencies. With docker each container can have its own dependencies, libraries, processes, networks, and mounts.

The purpose of docker is to package and containerise applications so we can ship them and run them as many times as we need.

Many products are containerised on Dockerhub repository. Such products include OS versions, database versions etc.

All containers share the same OS kernel.

What is an image?

A template used to create containers

What are containers?

Running instances of containers that are isolated and have their own processes

How long does a container live?

As long as the process inside it