Docker

## Introduction

This document covers

Introduction

### 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.

## Docker commands

### General

|  |  |
| --- | --- |
|  | Column Header |
| Version | docker --version |
| Pull image from Dockerhub | docker pull <image-name> |
| Start container from image | docker run <image-name> |
| List all running containers | docker ps |
| Stop container | docker stop <contId> |
| List all containers including stopped | docker ps -a |
| Delete container | docker rm <container-id> |
| Image List | docker images |
| Delete image | docker rmi <image-name> |
| Map container port to host port | docker run -p <host-port>:<container-port> myapp |
| Map container directory to host directory | docker run -v <host-dir> <cont-dir> |
| Inspect running container | docker inspect <image-name> |
| Execute command on container |  |

### Run

|  |  |
| --- | --- |
|  | Column Header |
| Start a container | docker run hello-world |
| Start container in background | docker run -d hello-world |
| Start container with std in/out | docker run -i -t <image> |

### Port mapping

The following runs a container and maps port 80 on the container to port **4000** on the container host.

docker run -d -p **4000**:80 --name myapp2 aspnetapp

### 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 logs

We can list the logs for the container using the following command.

docker logs <container-id>

The following shows a simple example.

> docker logs 6ff

> Hosting environment: Production

> Content root path: /app/

> Now listening on: http://[::]:80

> Application started. Press Ctrl+C to shut down.

### Docker Inspect

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

docker inspect <container-id>

### Viewing the directory structure of running container

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.

## Dockerize ASP.Net core

We will Dockerize a simple ASP.NET core server to familiarize ourselves with how we can use docker. To start with we setup a super simple end point.

### 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();

}

}

And a simple launchSettings.json as follows.

{

"profiles": {

"ASPHelloWorld": {

"commandName": "Project",

"launchBrowser": true,

"applicationUrl": "http://localhost:5000",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

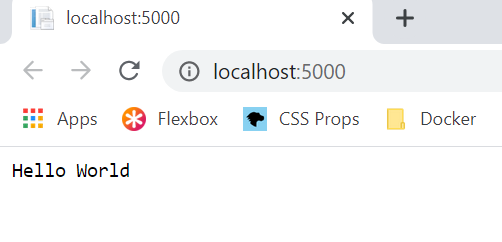
}

}

}

}

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



### 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>

#### Create the Docker File

##### 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 to refer to this stage.

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 container 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 run to use 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 two steps start up 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 ./

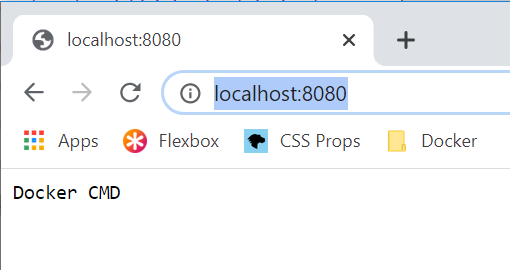
ENTRYPOINT ["dotnet", "ASPHelloWorld.dll"]

CMD ["Docker CMD"]

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.

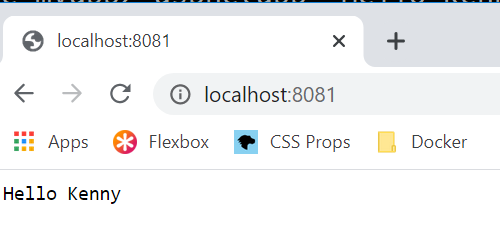


<https://stackoverflow.com/questions/48669548/why-does-aspnet-core-start-on-port-80-from-within-docker>

##### 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"



### Mapping directory from docker container to host.

Now consider the situation where our application logs to the container via Serilog. The configuration is as follows.

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"]

CMD ["Docker CMD"]

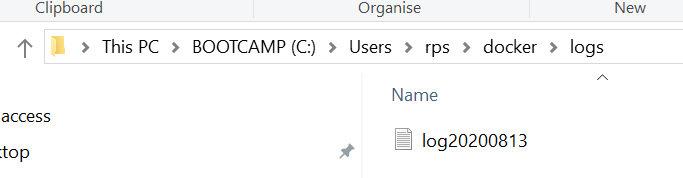
We now build the container.

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\rps\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



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.

## 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"

## Docker Theory

### Overview of Docker

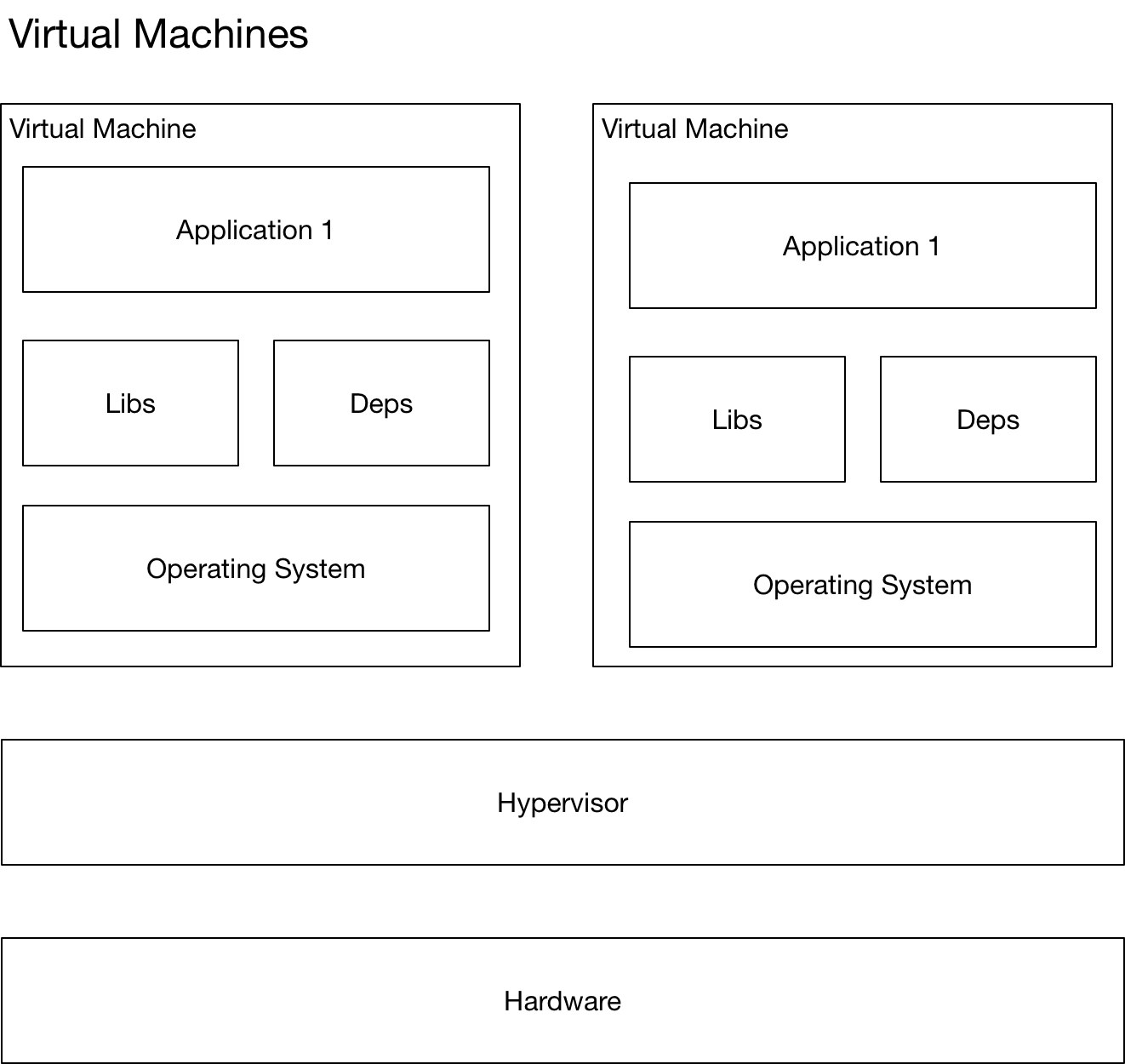
Docker runs on the OS and manages multiple containers. Each container has its own set of libraries and other dependencies. These dependencies are used by the application that runs inside the container.

A docker image is a template that is used to create docker containers. Each container is hence an instance of the image. A container is created using the docker run command. Dockerhub is a public repository of images for applications such as MongoDB and Node.js. We can also create our own images.

A container is a running process with some added features to keep it isolated from the host and from other containers. Each container has its own private filesystem. The filesystem is provided by the docker image.

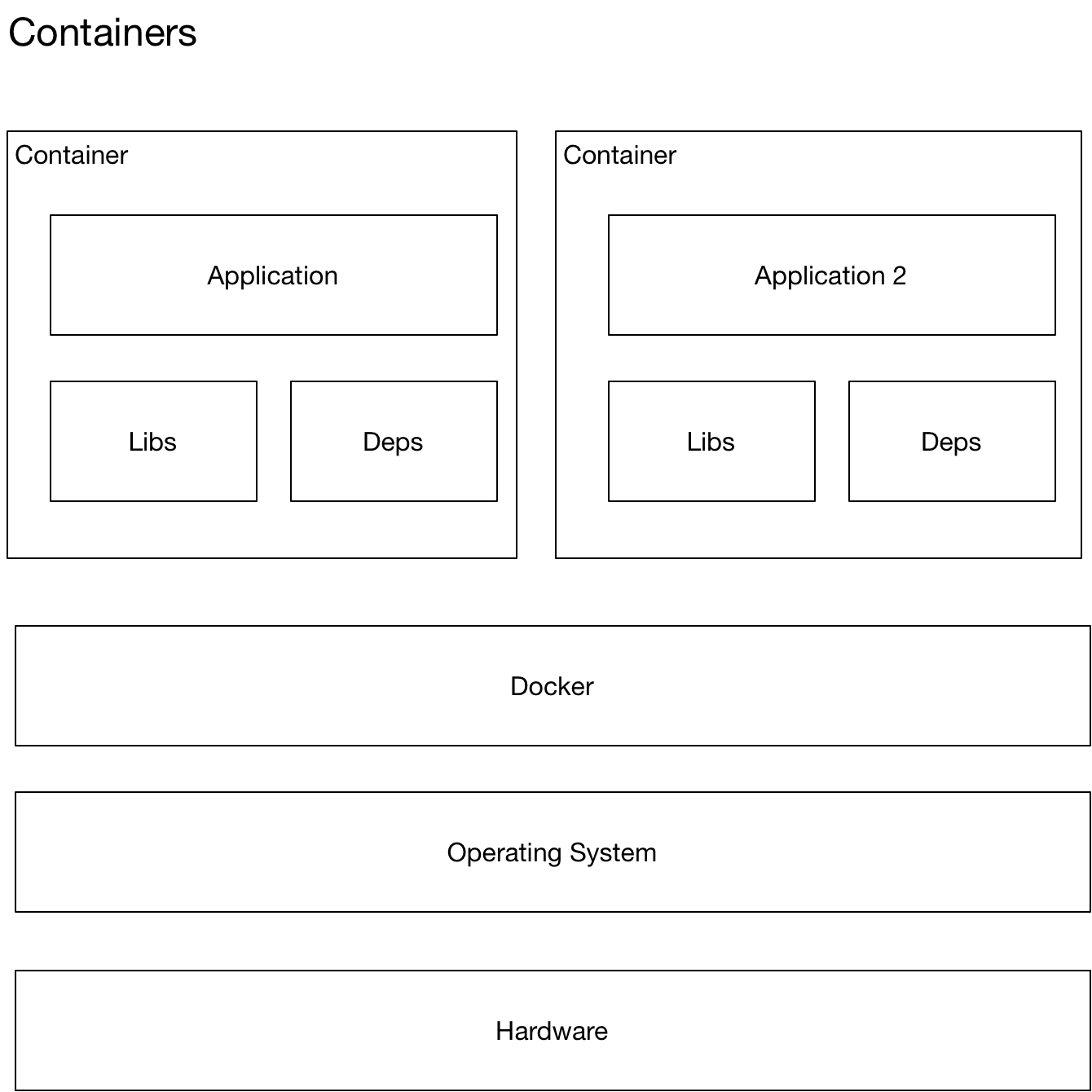
#### Docker vs Virtual Machines

Consider the following diagram that shows how virtualization works.



* Multiple virtual machines running on the same hardware.
* Heavyweight solution
* Each virtual machine has its own operating system.
* Each VM typically requires GB of desk space as it runs whole OS.
* Complete Isolation
* High overhead
* Start-up time can be minutes as OS needs to start.

Now let us look at Docker.



* Lightweight solution
* Each container typically MB in size
* Starts up in second.
* Docker has less isolation as more resources are shared across containers (kernel)

### 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 run image:tag | Docker run redis: | Show the version of docker installed |

## 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