Dockerfile and Docker Compose

.NET

docker compose *is a tool for defining and running multi-container Docker applications.* ***docker compose*** *uses a YAML file to configure an application's services.*

HTTPS://DOCS.DOCKER.COM/COMPOSE/

Dockerfile

https://docs.docker.com/engine/reference/builder/
https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

A ***Dockerfile*** is a text document that contains all the commands a user would call on the command line to assemble a Docker ***image***.

The ***Dockerfile*** is used by the docker build command to create a container ***image***.

docker build executes the instructions from the dockerfile in order.

```
$ docker build -t svendowideit/ambassador .

Sending build context to Docker daemon 15.36 kB
Step 1/4 : FROM alpine:3.2
 ---> 31f630c65071
Step 2/4 : MAINTAINER SvenDowideit@home.org.au
 ---> Using cache
 ---> 2a1c91448f5f
Step 3/4 : RUN apk update && apk add socat && rm -r /var/cache/
 ---> Using cache
 ---> 21ed6e7fbb73
Step 4/4 : CMD env | grep _TCP= | (sed 's/.*_PORT_\([0-
9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat -t 100000000 TCP4-
LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/' && echo wait) | sh
 ---> Using cache
 ---> 7ea8aef582cc
Successfully built 7ea8aef582cc
```

Dockerfile Format

https://docs.docker.com/engine/reference/builder/#format
https://docs.docker.com/engine/reference/builder/#official-releases

---

The ***Dockerfile*** follows a format very similar to a bash script. Instructions are run sequentially. By convention, keywords are uppercase. Comments begin with # and must be on their own line.

A ***Dockerfile*** must begin with a FROM instruction. FROM specifies the Parent Image from which you are building.

FROM may only be preceded by (one or more) ARG instructions which declare arguments that are used in FROM lines in the ***Dockerfile***.

Pre-made ***images*** for building ***Dockerfiles*** are available in the docker/dockerfile repo on Docker Hub (https://hub.docker.com/).

```
FROM alpine:3.2
#this is a commentRUN apk update && apk add socat && rm
–r /var/cache/
CMD env | grep _TCP= | (sed 's/.*_PORT_\([0-
9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat -t 100000000 TCP4-
LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/' && echo wait) | sh
```

ENV(ironment) variables

https://docs.docker.com/engine/reference/builder/#environment-replacement

The ENV instruction sets an environment variable <key> to <value>. The value will be in the environment for all subsequent instructions.

Environment variables set using ENV will persist when a container is run from the resulting image. docker inspect allows you to view them. You can change them with docker run --env <key>=<value>.

The ENV instruction has two forms.

- ENV <key> <value> sets a single variable to a value.
- ENV <key>=<value> ..., allows for multiple variables to be set at one time. Quotes and backslashes can be used to include (escape) spaces within values.

```
ENV <key> <value>
ENV <key>=<value> ...
```

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

yields the same results as

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

Dockerfile commands – FROM

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

A ***Dockerfile*** must start with FROM. FROM initializes a new build stage and sets the ***Base Image*** for subsequent instructions. ARG is the only instruction that may precede FROM in the ***Dockerfile***.

FROM can appear multiple times within a single ***Dockerfile*** to create multiple images or use a build stage as a dependency for another.

Give a name to a new build stage by adding AS to the FROM instruction. The name can be used in subsequent FROM. COPY --from=<name|index> instructions to refer to the image built in the previous stage.

FROM can use variables that are declared by any ARGs occurring before the first FROM.

An ARG declared before FROM is outside of a build stage, so it can't be used in any instruction after a FROM. Only the FROM itself can use it.

```
FROM [--platform=<platform>] <image> [AS <name>]
```

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app

FROM extras:${CODE_VERSION}
CMD  /code/run-extras
```

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

Dockerfile commands – WORKDIR

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

WORKDIR sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile. If a WORKDIR doesn't exist, it will be automatically assigned.

WORKDIR can be used multiple times in a ***Dockerfile***. When a path to a file is provided, it must be relative to the path of the previous WORKDIR.

WORKDIR can use environment variables previously set using ENV. Only environment variables explicitly set in the ***Dockerfile*** can be used.

```
WORKDIR /path/to/workdir
```

The output of pwd (path to working directory) in this Dockerfile will be /a/b/c

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

The output of pwd will be /path/$DIRNAME

Dockerfile commands – RUN

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

RUN has 2 forms:

- RUN <command> (shell form, the command is run in a shell)
- RUN ["executable", "param1", "param2"] (*exec* form)

RUN will execute commands in a new layer on top of the current image and then commit the results. The resulting committed image will be used for the next ***Dockerfile*** step.

Layering RUN instructions and generating commits conforms to the core concepts of ***Docker*** where containers can be created from any point in an image's history.

'exec' form syntax makes it possible to RUN commands using a ***base image*** that does not contain the specified shell executable.

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'
```

```
RUN ["/bin/bash", "-c", "echo hello"]
```

Exec form syntax

Dockerfile commands – COPY

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

The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the ***Container*** at the path <dest>.

```
COPY [--chown=<user>:<group>] <src>... <dest>
COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

Multiple <src> resources may be specified. Paths are interpreted relative to the ***context***. <src> may also contain wildcards.

COPY has two forms:

- COPY [--chown=<user>:<group>] <src>... <dest>
- COPY [--chown=<user>:<group>] ["<src>",... "<dest>"] (used for whitespace)

```
COPY hom* /mydir/
```

adds all files starting with "hom…".

```
COPY test.txt relativeDir/
```

adds "test.txt" to <WORKDIR>/relativeDir/

The (optional) –chown flag specifies a given userName, groupName, or UID/GID combination to request specific ownership of the copied content.

```
COPY --chown=55:mygroup files* /somedir/
```

If the user or group are invalid, the build fails on COPY.

Dockerfile commands – EXPOSE

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

EXPOSE informs Docker that the container listens on the specified ports. By default, the port listens on TCP.

EXPOSE does not actually publish the port. It functions as documentation between those who build the image and those who run the container.

You can override ***Dockerfile*** EXPOSE settings at runtime with the -p flag.

To publish the port when running the container, use the -p flag on docker run to publish and map one or more ports, or the -P flag to publish all exposed ports and map them to high-order ports.

```
EXPOSE <port> [<port>/<protocol>...]
```

```
EXPOSE 80/udp
```

```
docker run -p 80:80/tcp -p 80:80/udp ...
```

To override EXPOSE settings

Dockerfile commands – CMD

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

CMD and ENTRYPOINT are very similar. They both provide startup commands.

CMD provides default startup commands for a container. These defaults can include an executable.

CMD does not execute anything at build time but specifies the intended command for the image. If different commands are entered at time of docker run, CMD commands are overridden.

The CMD instruction has three forms:

- CMD ["executable","param1","param2"] (exec form. Preferred)
- CMD ["param1","param2"] (default params to ENTRYPOINT)
- CMD command param1 param2 (shell form)

There can only be one CMD instruction in a ***Dockerfile***.

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

This ***exec*** form is the preferred format of CMD. Params must be expressed as strings in the array.

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

Shell form. The <command> will execute in /bin/sh -c

Dockerfile commands – ENTRYPOINT (1/2)

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

ENTRYPOINT allows configuration of a container to run as an executable. It has two forms:

- ENTRYPOINT ["executable", "param1", "param2"] (*exec* form: Preferred)
- ENTRYPOINT command param1 param2 (shell form*)

ENTRYPOINT must be specified when CMD omits the executable.

If CMD is used to provide default arguments for ENTRYPOINT. Both the CMD and ENTRYPOINT instructions should use the JSON array format.

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

```
docker run -i -t --rm -p 80:80 nginx
```

This command starts nginx with its default content, listening on port 80:

*Click the link above for more specifics on the shell form of ENTRYPOINT instruction.

Dockerfile commands – ENTRYPOINT(2/2)

https://docs.docker.com/engine/reference/builder/#exec-form-entrypoint-example

Command line arguments to docker run <image> will be appended after all elements in an ***exec*** form ENTRYPOINT. They override all elements specified using CMD. This allows docker run <image> -d to pass the -d argument to the entry point. You can override the ENTRYPOINT instruction by using the --entrypoint flag at docker run time.

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

Use the ***exec*** form of ENTRYPOINT to set default commands or arguments and then use CMD to set additional defaults that are more likely to be changed on execution.

Configure the above, then run the container to see that 'top' is the only process.

```
$ docker run -it --rm --name test  top -H

top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352 buffers
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    1 root      20   0   19744   2336   2080 R  0.0  0.1   0:00.04 top
```

*Click the link above for more specifics on the shell form of ENTRYPOINT instruction.

Dockerfile commands – LABEL

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

LABEL adds metadata to an image as a key-value pair.

- Include spaces by using quotes (or backslashes) as in command-line parsing.
- You can specify multiple labels on a single line.
- Labels included in parent images (images in the FROM line) are inherited.
- If a LABEL already exists with a different value, it's overwritten
- To view an image's labels, use docker image inspect command.

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

```
docker image inspect --format='' myimage
```

```
{
  "com.example.vendor": "ACME Incorporated",
  "com.example.label-with-value": "foo",
  "version": "1.0",
  "description": "This text illustrates that label-values can span multiple lines.",
  "multi.label1": "value1",
  "multi.label2": "value2",
  "other": "value3"
}
```

Dockerfile commands – ADD

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

ADD <src> <dest> copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

- Multiple <src> resources may be specified.
- File or directory paths are written relative to the source of the build context.
- Each <src> may contain wildcards. Matching is done using Go's filepath.Match rules.

There are many more options and configurations for ADD in the docs linked above.

use a relative path to add "test.txt" to <WORKDIR>/relativeDir/

```
ADD test.txt relativeDir/
```

use an absolute path to add "test.txt" to /absoluteDir/

```
ADD test.txt /absoluteDir/
```

Escaping special chars. Adds "test.txt" to <WORKDIR>/relativeDir/

```
ADD arr[[]0].txt /mydir/
```

Dockerfile commands – VOLUME

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

VOLUME creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

The exact location of the volume on the host machine is decided by the docker engine. The volume is accessed by the docker engine using the specified name given at creation.

The value can be a JSON array, (VOLUME ["/var/log/"]), a plain string (VOLUME /var/log) or have multiple arguments (VOLUME /var/log /var/db).

docker run initializes the newly created volume with any data that exists at the specified location within the base image.

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

The Dockerfile example above results in an image that causes docker run to create a new mount point at /myvol and copy the greeting file into the newly created volume.

Dockerfile commands – USER

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

USER sets the user's name or UID and the user group or GID to use when running the image for any RUN, CMD and ENTRYPOINT instructions that follow it in the ***Dockerfile***.

```
USER <user>[:<group>]
```

```
USER <UID>[:<GID>]
```

The user will have only the specified group membership. Any other configured group memberships will be ignored.

On Windows, the user must be created first if it's not a built-in account. This is done with net user called as part of a ***Dockerfile***.

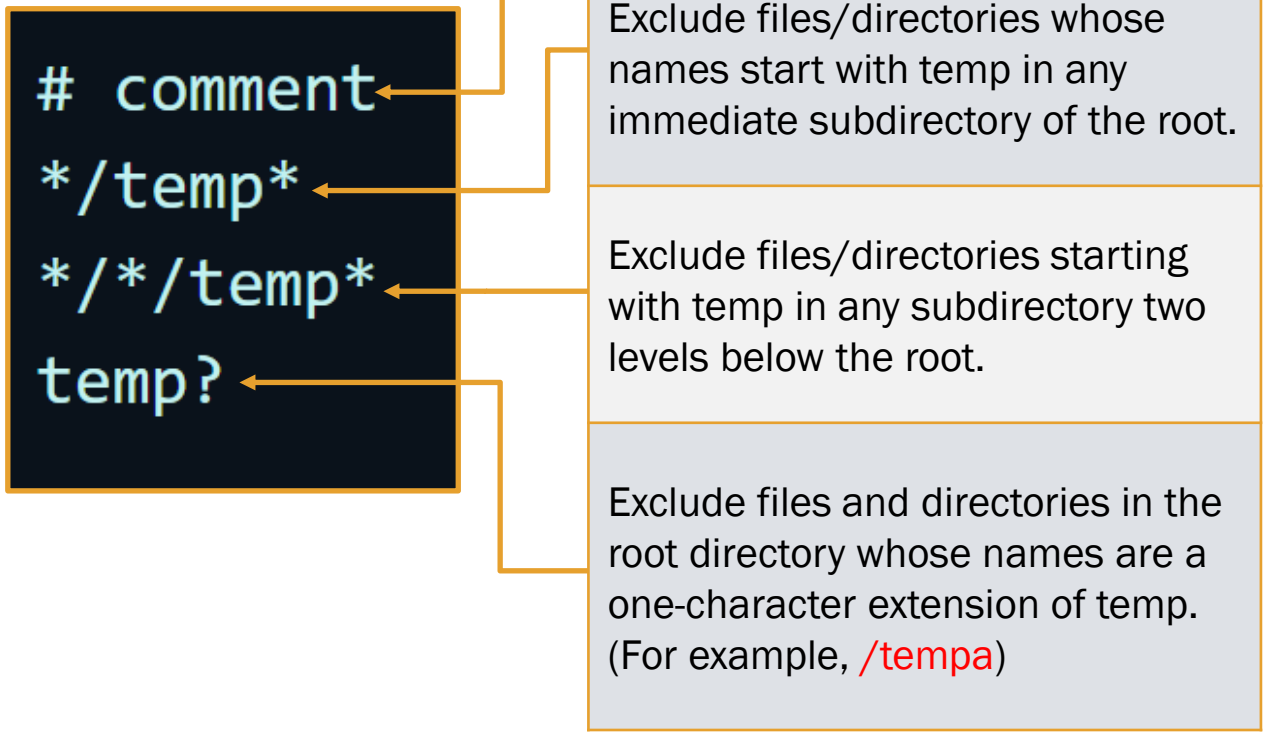
```
FROM microsoft/windowsservercore
# Create Windows user in the container
RUN net user /add patrick
# Set it for subsequent commands
USER patrick
```

.dockerignore file

https://docs.docker.com/engine/reference/builder/#dockerignore-file

a .dockerignore file allows you to exclude files and directories from docker build.

The .dockerignore file is a newline-separated list of filenames relative to the root directory of the ***context***. The root of the ***context*** is also the working directory.



docker compose – Overview

https://docs.docker.com/compose/

docker-compose up allows you to create and start all the services from your configuration. docker-compose works in all environments: production, staging, development, testing, and CI/CD workflows.

Using docker compose is a three-step process:

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 the ***Compose*** app starts and runs your entire app.

```
version: '2.0'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Docker Compose - Features

https://docs.docker.com/compose/#features

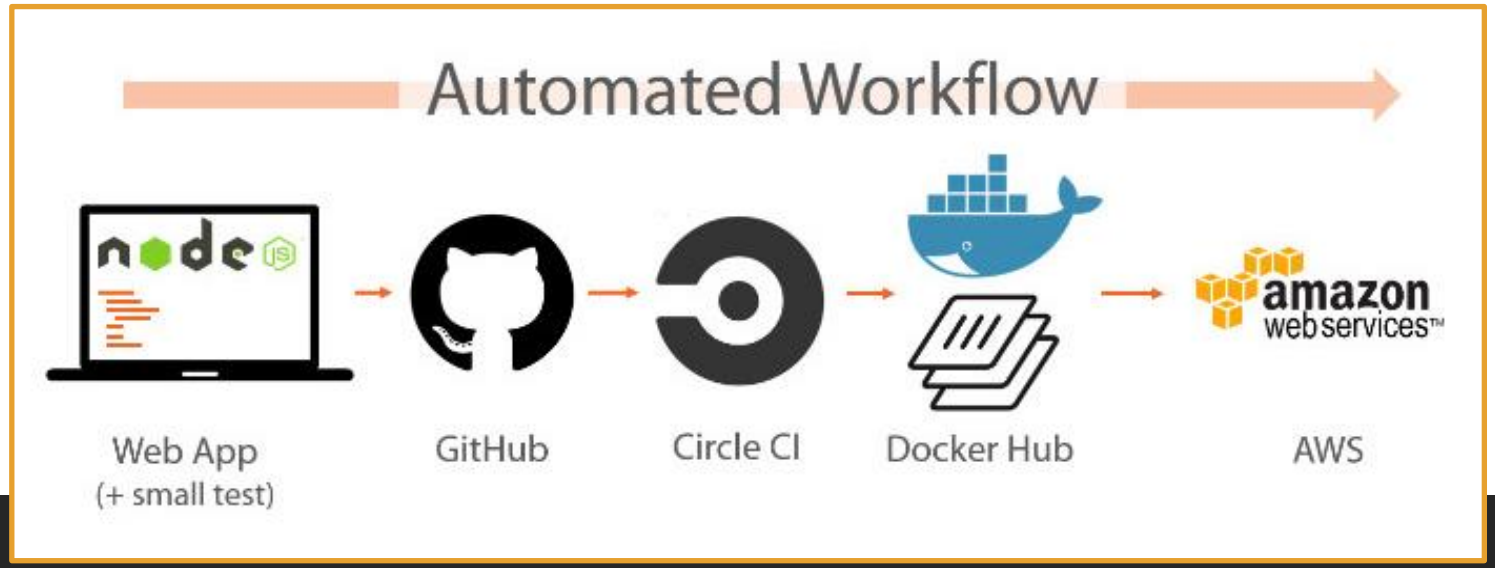
Have many isolated environments on one host	Preserve volume data when containers are created	Recreate containers that have changed	A composition is portable between environments
Because ***Compose*** uses unique names for each project, the projects run independently of each other. Use –p to give your project a unique name.	When docker-compose up runs, it copies the volumes from the old ***container*** to the new ***container***. This ensures that any data you've created in volumes isn't lost.	***Compose*** re-uses existing ***containers*** if they haven't been changed.	You can use variables in the ***Compose*** file to customize your composition for different environments or different users.

Docker Compose – Use Cases

https://docs.docker.com/compose/#common-use-cases

Compose is used Extensively with ***Continuous Deployment/Continuous Integration (CI/CD)***.

Compose is used in a development environment. A docker-compose.yml file is used to automate testing configurations and dependencies so that workflows are automated and end-to-end testing is faster and easier.

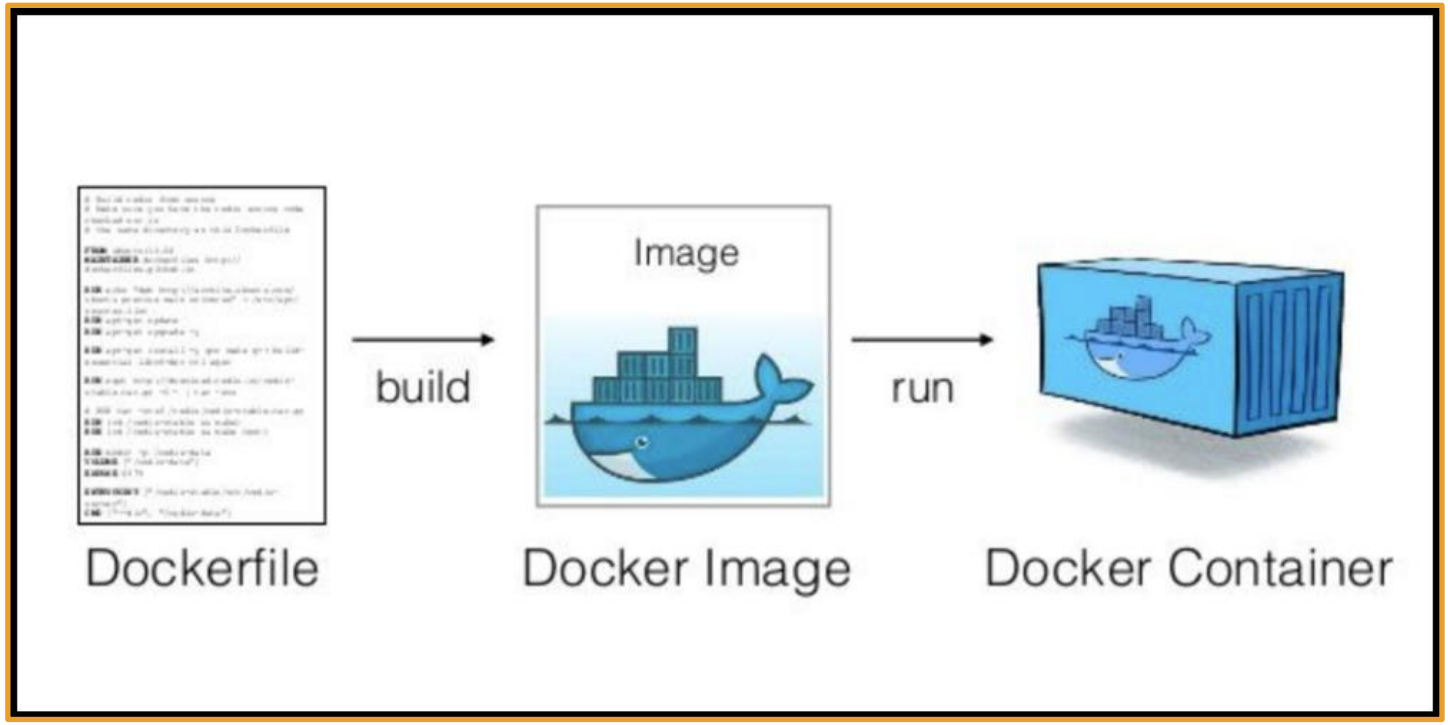


Resource

https://medium.com/@saurabh.singh0829/how-to-build-application-inside-and-outside-docker-dockerfile-structure-and-commands-f542b58cd830

How to build an application inside and outside of Docker.

Dockerfile structure and commands.



Sample ASP.NET Core App with SQL Server – Step-by-Step

- Install Docker Desktop (includes Docker Engine, Docker CLI client, Docker Compose, Notary, Kubernetes, and Credential Helper.)
- Create a new directory for your application. This is the 'context' of the project.
- Docker doesn't work on versions earlier that Windows 10.

- Virtual box won't work on Windows. (9 min mark in Edureka video)
- Run windows powershell as administrator
- Use docker –version to see what you have.
- docker run hello-world => downloads the image automatically.
- Docker pull ubuntu – to pull the official ubuntu image.
- Docker run –it –d ubuntu => run the ubuntu image "detached" and create a container
- EXIT to exit the container
- Docker commit [containerNum] accountName/imageName

https://docs.microsoft.com/en-us/dotnet/core/docker/build-container

---

Tutorial: Containerize a .NET Core app

Make and containerize a .NET Core App with SQ: Server (below).

https://docs.docker.com/compose/aspnet-mssql-compose/

https://docs.docker.com/compose/gettingstarted/