From Zero to Docker

Training | 2019.05.16 | Mário Dagot, Jorge Dias

Docker is an open platform for developing, shipping, and running applications. Through the course of this training we will guide you to the most common feature and use cases of docker. Take this as an introduction and an opportunity to dive into the docker world.

AGENDA

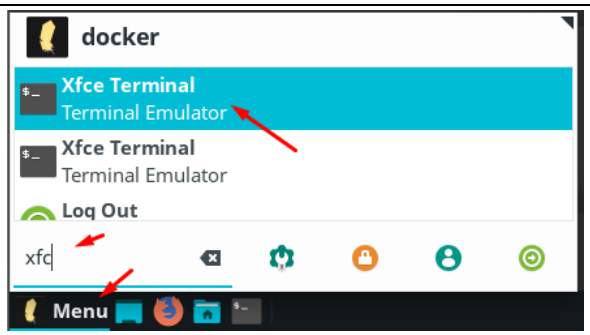
- 01 - Install Vim and Terminator and VSCode
- 02 - Install Docker CE for Ubuntu
- 03 - Hello from Busybox
- 04 - Webapp with Docker
- 05a - Webapp with Docker - My first Dockerfile – Nginx
- **05b.1 - Webapp with Docker - My first Dockerfile - Dotnet Core**
- 05b.2 - Webapp with Docker - My first Dockerfile MultiStage - Dotnet Core
- 06 - Save and Restore and Push to Docker Hub
- 07a - Webapp with database integration - My first network – SpringBoot
- 07b - Webapp with database integration - My first docker-compose – SpringBoot

05B.1 - WEBAPP WITH DOCKER - MY FIRST DOCKERFILE - DOTNET CORE

Objective

- Pull the dotnet core sdk from docker hub and use it to create, build and run a c# web application
- Check the logs of containers
- Create a docker image and package our C# web application
- Run the docker packaged C# web application

Step by Step

Open a terminal windows	

Let's pull the dotnet core sdk from docker hub. We can run the dotnet command, from inside the image, and see that it is there. HINT: If you run a docker image and if it is not yet on your system it will automatically pull it from docker hub.	docker ~ docker pull mcr.microsoft.com/dotnet/core/sdk:2.2 2.2: Pulling from dotnet/core/sdk c5e155d5a1d1: Pull complete 221d80d00ae9: Pull complete 4250b3117dca: Pull complete 3b7ca19181b2: Pull complete e27e0cfb92ac: Pull complete 0321dc759d89: Pull complete 5ece62b84f45: Pull complete Digest: sha256:c250db2834992273f58a4d135987969523c2420e3ce8311424fa73392947c694 docker ~ docker run --rm mcr.microsoft.com/dotnet/core/sdk dotnet Usage: dotnet [options] Usage: dotnet [path-to-application] Options: -h|--help Display help. --info Display .NET Core information. --list-sdks Display the installed SDKs. --list-runtimes Display the installed runtimes. path-to-application: The path to an application .dll file to execute. Status: Downloaded newer image for mcr.microsoft.com/dotnet/core/sdk:2.2
We start by creating our workspace folder and then run the basic dotnet command to create our base dotnet core web application. As we can see a folder has been created and it contains C# code. What was all those extra flags? --rm we already know. Remove the container once it finishes.	docker ~ mkdir ~/workspace docker ~ cd ~/workspace/ docker ~ workspace ls -l total 0 docker ~ workspace docker run --rm -v /home/docker/workspace/:/app mcr.microsoft.com/dotnet/core/sdk dotnet new webapi --name myapp -o /app/myapp Getting ready... The template "ASP.NET Core Web API" was created successfully. Processing post-creation actions...

--name we use it to define the name the container.
-v mounts a specific folder on the host machine onto the container.
-o sets the working dir inside the container. In practical terms when the dotnet new webapi executes it will execute on the folder /app/myapp
 	
Finally, dotnet new webapi creates a template web application in C# so that we can play around.
 	
Notice that everything was built by user root. The container has its own set of users that may not match the hosts (with the exception of root that has always uid 0 in linux). This is something to keep in mind.
	`Running 'dotnet restore' on /app/myapp/myapp.csproj...`
	`  Restore completed in 1.18 sec for /app/myapp/myapp.csproj.`
	
	`Restore succeeded.`
	
	`docker ~ workspace ls -l`
	`total 4`
	`drwxr-xr-x 5 root root 4096 mai  8 18:37 myapp`
	`docker ~ workspace ls -l myapp/`
	`total 32`
	`-rw-r--r-- 1 root root  146 mai  8 18:37 appsettings.Development.json`
	`-rw-r--r-- 1 root root  105 mai  8 18:37 appsettings.json`
	`drwxr-xr-x 2 root root 4096 mai  8 18:37 Controllers`
	`-rw-r--r-- 1 root root  412 mai  8 18:37 myapp.csproj`
	`drwxrwxrwx 2 root root 4096 mai  8 18:37 obj`
	`-rw-r--r-- 1 root root  627 mai  8 18:37 Program.cs`
	`drwxr-xr-x 2 root root 4096 mai  8 18:37 Properties`
	`-rw-r--r-- 1 root root 1568 mai  8 18:37 Startup.cs`

Let's build the application.	`docker ~ workspace docker run --rm --workdir /app/myapp -v /home/docker/workspace/:/app mcr.microsoft.com/dotnet/core/sdk dotnet restore`
	`  Restore completed in 65.78 ms for /app/myapp/myapp.csproj.`
	`docker ~ workspace docker run --rm --workdir /app/myapp -v /home/docker/workspace/:/app mcr.microsoft.com/dotnet/core/sdk dotnet build`
	`Microsoft (R) Build Engine version 16.0.450+ga8dc7f1d34 for .NET Core`
	`Copyright (C) Microsoft Corporation. All rights reserved.`
	
	`  Restore completed in 104.48 ms for /app/myapp/myapp.csproj.`
	`  myapp -> /app/myapp/bin/Debug/netcoreapp2.2/myapp.dll`
	
	`Build succeeded.`
	`  0 Warning(s)`
	`  0 Error(s)`
	`Time Elapsed 00:00:04.18`

And then run it. Note we exposed the port 80 from to container to 5000 on the host.	`docker ~ workspace docker run -d -p 5000:80 --workdir /app/myapp -v /home/docker/workspace/:/app mcr.microsoft.com/dotnet/core/sdk dotnet run --urls http://0.0.0.0:80` `3cc1378bf125776623e002bfbee522e6f37a8e6a9e9fc6405b8b5f56dc1a3a00`

Nothing we have never seen before. Keep in mind, we never installed the C# SDK. And we can also easily replace the version 2.2 by another. Or even keep both without any conflicts on our system.	
Let's check the logs. By default all stdout from the container can be seen using the docker logs command.	docker ~ > workspace > docker logs hungry_minsky info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0] User profile is available. Using '/root/.aspnet/DataProtection-Keys' as key repository; keys will not be encrypted at rest. info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[58] Creating key {b3d811c7-b801-4c7e-b046-8f8db37ae0b6} with creation date 2019-05-08 17:54:56Z, activation date 2019-05-08 17:54:56Z, and expiration date 2019-08-06 17:54:56Z. warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35] No XML encryptor configured. Key {b3d811c7-b801-4c7e-b046-8f8db37ae0b6} may be persisted to storage in unencrypted form. info: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[39] Writing data to file '/root/.aspnet/DataProtection-Keys/key-b3d811c7-b801-4c7e-b046-8f8db37ae0b6.xml'. Hosting environment: Development Content root path: /app/myapp Now listening on: http://0.0.0.0:80 Application started. Press Ctrl+C to shut down.
Let's call the service. First we do an internal call from inside the container. And then from the host using the exposed port. Do try accessing the two URLs from your browser and see what happens.	docker ~ > workspace > docker exec -it hungry_minsky curl http://localhost:80/api/values ["value1","value2"] docker ~ > workspace > curl http://localhost:5000/api/values ["value1","value2"]
And now the fun stuff: our first Dockerfile. Dockerfile allow us to create the docker image in a standard way. It can also be reproduced every time	docker ~ > workspace > cd myapp/ docker ~ > workspace > myapp > sudo vi Dockerfile docker ~ > workspace > myapp > cat Dockerfile FROM mcr.microsoft.com/dotnet/core/sdk:2.2

with the same result and in an automated fashion.	`WORKDIR /app`
	`COPY . /app`
	
	`RUN dotnet restore && \`
	`  dotnet build && \`
	`  dotnet publish -o ./publish -c Release`
	
	`ENTRYPOINT ["dotnet", "/app/publish/myapp.dll"]`

We can now build the image using docker build command.
 	
We used the -t to define the name and tag of the image. We called it myapp.
	`docker  ~  workspace  myapp  docker build -t myapp .`
	`Sending build context to Docker daemon  3.803MB`
	`Step 1/5 : FROM mcr.microsoft.com/dotnet/core/sdk:2.2`
	`  ---> 61da26769572`
	`Step 2/5 : WORKDIR /app`
	`  ---> Using cache`
	`  ---> 88b0779c5fc4`
	`Step 3/5 : COPY . /app`
	`  ---> 8579dfcecdc0`
	`Step 4/5 : RUN dotnet restore &&  dotnet build &&  dotnet publish -o ./publish -c Release`
	`  ---> Running in 3c129970fc66`
	`  Restore completed in 21.48 sec for /app/myapp.csproj.`
	`Microsoft (R) Build Engine version 16.0.450+ga8dc7f1d34 for .NET Core`
	`Copyright (C) Microsoft Corporation. All rights reserved.`
	
	`  Restore completed in 56 ms for /app/myapp.csproj.`
	`  myapp -> /app/bin/Debug/netcoreapp2.2/myapp.dll`
	
	`Build succeeded.`
	`  0 Warning(s)`
	`  0 Error(s)`
	
	`Time Elapsed 00:00:03.59`
	`Microsoft (R) Build Engine version 16.0.450+ga8dc7f1d34 for .NET Core`
	`Copyright (C) Microsoft Corporation. All rights reserved.`
	
	`  Restore completed in 96.49 ms for /app/myapp.csproj.`

	`  myapp -> /app/bin/Release/netcoreapp2.2/myapp.dll`
	`  myapp -> /app/publish/`
	`Removing intermediate container 3c129970fc66`
	` ---> f34d77571573`
	`Step 5/5 : ENTRYPOINT ["dotnet", "/app/publish/myapp.dll"]`
	` ---> Running in a8402be667ad`
	`Removing intermediate container a8402be667ad`
	` ---> 0b8583945316`
	`Successfully built 0b8583945316`
	`Successfully tagged myapp:latest`

We can do it again. Compare the two outputs.
 	
We can see in some of the commands the string “Using cache”.
 	
The docker engine is smart enough and uses something called the Union Filesystem. Every command (well, not every, all that affect the filesystem) will result in a new layer. This allows us to easily reuse layers and speed up the build and final image size.
	`docker  ~  workspace  myapp  docker build -t myapp .`
	`Sending build context to Docker daemon  3.803MB`
	`Step 1/5 : FROM mcr.microsoft.com/dotnet/core/sdk:2.2`
	` ---> 61da26769572`
	`Step 2/5 : WORKDIR /app`
	` ---> Using cache`
	` ---> 88b0779c5fc4`
	`Step 3/5 : COPY . /app`
	` ---> 8579dfcecdc0`
	`Step 4/5 : RUN dotnet restore &&  dotnet build &&  dotnet publish -o ./publish -c Release`
	` ---> Running in 3c129970fc66`
	`  Restore completed in 21.48 sec for /app/myapp.csproj.`
	`Microsoft (R) Build Engine version 16.0.450+ga8dc7f1d34 for .NET Core`
	`Copyright (C) Microsoft Corporation. All rights reserved.`
	
	`  Restore completed in 56 ms for /app/myapp.csproj.`
	`  myapp -> /app/bin/Debug/netcoreapp2.2/myapp.dll`
	
	`Build succeeded.`
	`  0 Warning(s)`
	`  0 Error(s)`
	
	`Time Elapsed 00:00:03.59`
	`Microsoft (R) Build Engine version 16.0.450+ga8dc7f1d34 for .NET Core`
	`Copyright (C) Microsoft Corporation. All rights reserved.`

```
  Restore completed in 96.49 ms for /app/myapp.csproj.
  myapp -> /app/bin/Release/netcoreapp2.2/myapp.dll
  myapp -> /app/publish/
Removing intermediate container 3c129970fc66
 ---> f34d77571573
Step 5/5 : ENTRYPOINT ["dotnet", "/app/publish/myapp.dll"]
 ---> Running in a8402be667ad
Removing intermediate container a8402be667ad
 ---> 0b8583945316
Successfully built 0b8583945316
Successfully tagged myapp:latest
```

If we look at our images we can see the new myapp image listed.

Well done.

```
docker  ~ > workspace > myapp  docker images
REPOSITORY                          TAG                 IMAGE ID
CREATED             SIZE
myapp                               latest              0b8583945316        2
minutes ago       1.78GB
<none>                              <none>              0169ca3a9398        4
minutes ago       1.78GB
<none>                              <none>              60539d0fba40        8
minutes ago       1.78GB
importedbusybox                     mytag               f2990efd90b3        9
hours ago         1.42MB
dagot/my-custom-nginx               v1                  67c388f94320        12
hours ago        109MB
my-custom-nginx                     latest              67c388f94320        12
hours ago        109MB
mcr.microsoft.com/dotnet/core/sdk   2.2                 61da26769572        13
hours ago        1.74GB
mcr.microsoft.com/dotnet/core/sdk   latest              61da26769572        13
hours ago        1.74GB
nginx                               latest              53f3fd8007f7        19
hours ago        109MB
busybox                             latest              af2f74c517aa        5
weeks ago         1.2MB
```

And we can just run it as standard image.

Curl it if you don't believe.

```
docker  ~ > workspace > myapp  docker run --rm -d -p 8080:80 --name myapp
myapp
904529fcdecf109430ae33002adc6e98590484d22b19f86858ada7131bf42c0c
docker  ~ > workspace > myapp
```

```
docker  ~ > workspace > myapp  curl http://localhost:8080/api/values
["value1","value2"] docker  ~ > workspace > myapp  docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                   NAMES
904529fcdecf        myapp               "dotnet /app/publish…"   15 seconds ago
Up 14 seconds       0.0.0.0:8080->80/tcp   myapp
docker  ~ > workspace > myapp  docker container rm -f myapp
```

Lessons learned

We learned how to use a pre made image, containing the C# SDK, to create, build and run a C# web application. From this experiment we can see how easy it would be to test the same web application using an older or newer version of dotnet core.

We learned how to automatize the process of creation of a docker image: the Dockerfile.

We learned how to package the C# web application in such a way that we can share the image with all its dependencies. Truly build once and run it everywhere (as long as the docker engine is there).

Revision History

Version	Date	Author	Description
1.0	2019.05.01	Mário Dagot, Jorge Dias	Initial Version