Search

Search

Docker Tutorials

Create first ASP.NET Core App in a Docker Container

ASP.NET Core APP with HTTPS in Docker

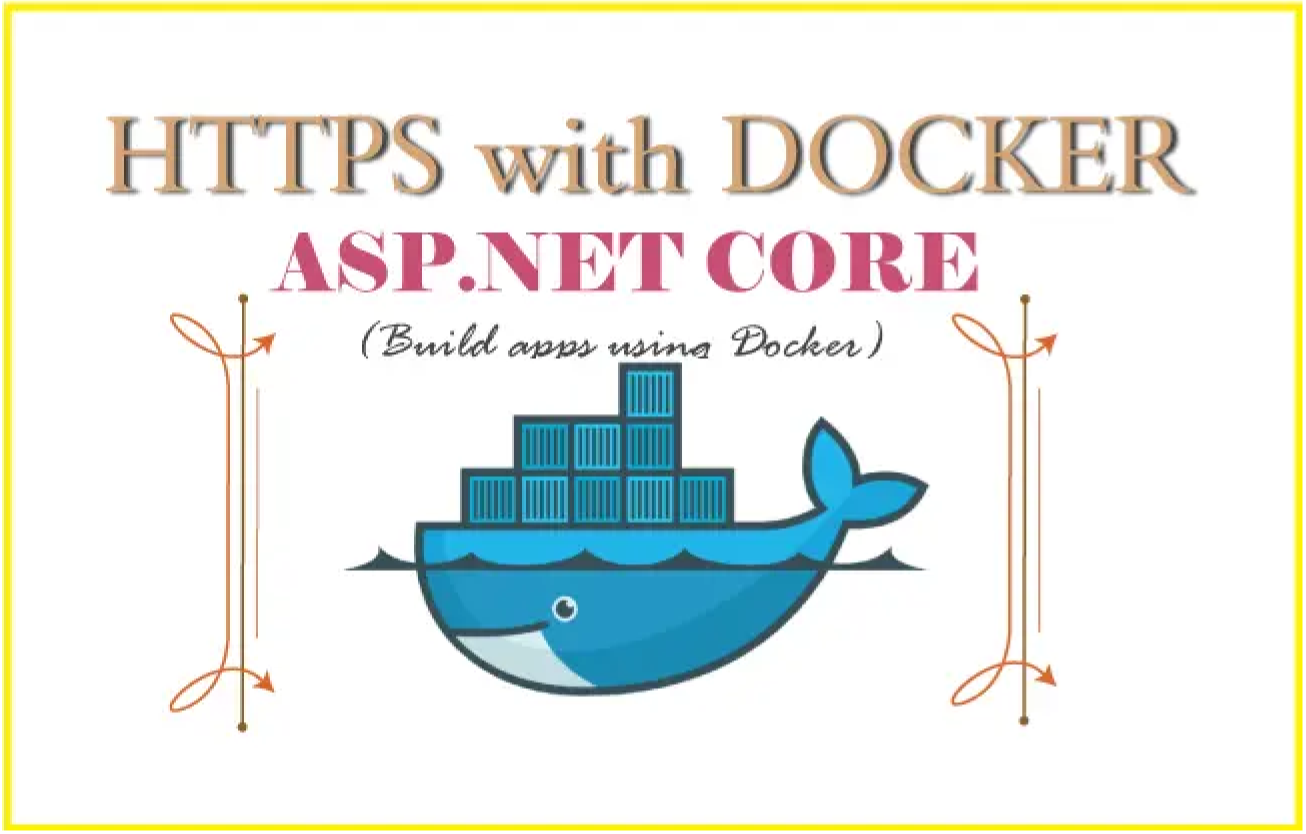
Deploy a Docker based ASP.NET Core app to Azure

Multi-Container ASP.NET Core App with Docker Compose

CRUD Operations in ASP.NET Core and SQL Server with Docker

ASP.NET Core APP with HTTPS in Docker

Last Updated: November 17, 2023



SSL Certificates are very necessary for the Trust, Identity and Encryption of an APP. In ASP.NET Core the apps use HTTPS Certificates by default, they use self-signed development certificates for development purpose. So, when you are hosting your app to a **Docker Container** then it is needed to tell docker where to find this development certificate in the machine. Once Docker knows the location of the HTTPS certificate then your app will start opening with https url, eg https://localhost:8001.

The procedure will be same for the production scenario also. So you can generate a free HTTPS certificate from Let's Encrypt, then tell your Docker app (which is running in Azure or AWS) to find the HTTPS certificate from a location.

Page Contents

ASP.NET Core Docker HTTPS Example

First create a new ASP.NET Core Web App (Model-View-Controller) in Visual Studio and name it DockerHttps, and make sure to select the option that says – Place solution and project in the same directory. After that add Dockerfile to the app.

Kindly note that you have to select Linux containers.

I covered Dockerfile and Docker commands for containers and images in great details on my tutorial ASP.NET Core Docker Example, kindly check it.

▼ This tutorial is a part of **ASP.NET Core apps on Docker** series.

This app is created on .NET 8. It's Dockerfile is shown below:

```
#See https://aka.ms/customizecontainer to learn how to customize your debug
container and how Visual Studio uses this Dockerfile to build your images for
faster debugging.

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER app
WORKDIR /app
EXPOSE 8080
EXPOSE 8081

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["DockerHttps.csproj", "."]
```

```
RUN dotnet restore "./././DockerHttps.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "./DockerHttps.csproj" -c $BUILD_CONFIGURATION -o /app/build

FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./DockerHttps.csproj" -c $BUILD_CONFIGURATION -o
/app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "DockerHttps.dll"]
```

Our ASP.NET Core app is ready to be hosted on **Docker with HTTPS Certificate**. You can close the Visual Studio now since we will now do all the work from the command prompt.

We now need to understand 2 important topics which are:

1. Environment variables
2. Docker Volume

ASP.NET Core Docker Environment Variables

Docker Environment Variables are used for configuring Docker Images and Docker Containers. The “-e” option in a docker command is used to set environment variables. I will use Environment Variables to configure a number of things, which are:

- HTTPS ports for the app
- HTTPS certificate password
- HTTPS certificate path

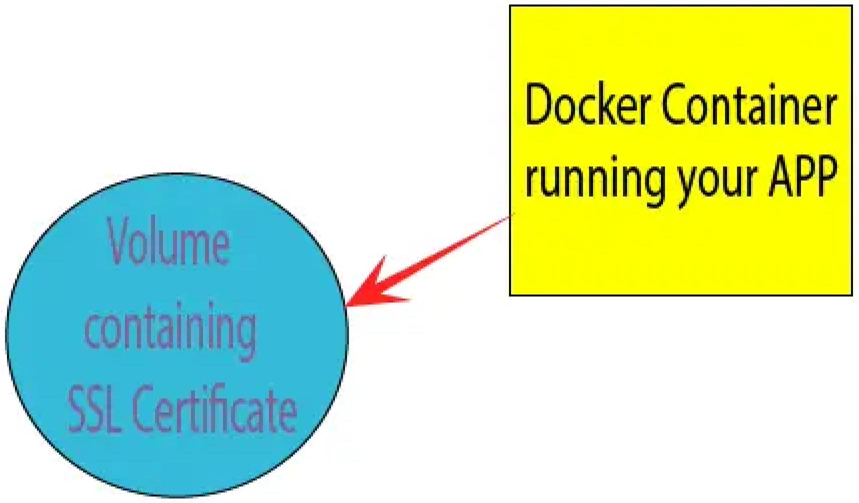
The certificate will remain outside the container and will be mapped to the container using Docker Volume.

ASP.NET Core Docker Volume

Suppose we want to access a file which contains some important password. The password is needed for the app which is running in a Docker Container. We want this file to remain accessible and never gets deleted even if the container is destroyed. How to do it?

The answer is through **Docker Volume**. The Docker Volume lives outside of the container in the file system of the hosting server. Docker will copy this volume files to inside of a container that needs them. So this means even if the container of our app gets destroyed in future due to some fault. Then docker will create a new container from the app image and copy files from volume to inside of this new container. Therefore the app will always keep running.

When running Docker Container in Azure, we can store important files in azure directory. The container can access these file from there. See the below image which explains this concept of **Docker Volume**.


Docker Container running your APP
Volume containing SSL Certificate

We can simply use the **Docker Volume** concept to store a SSL certificate in a volume, and then let our app, which is running in a docker container, to use it from there.

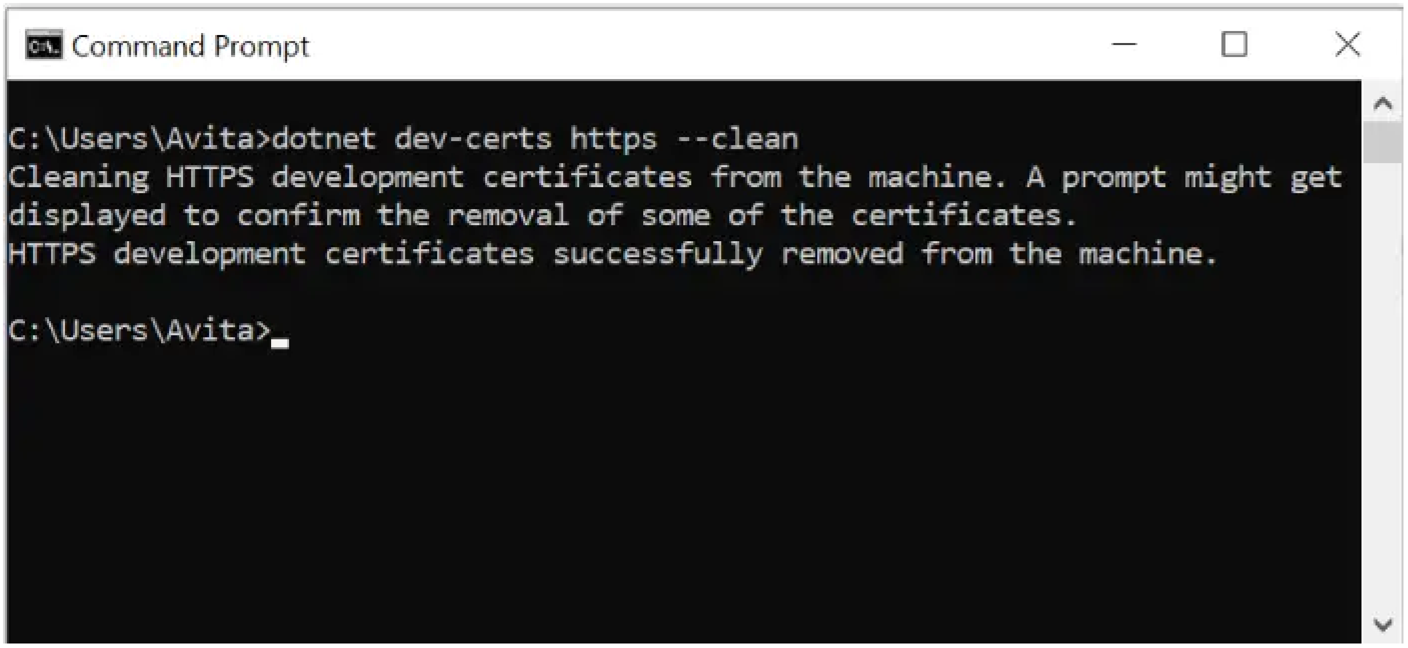
Use **-v** option in docker command to work with volumes.

Creating SSL with dotnet dev-certs

The dotnet dev-certs tool is used to create self-signed development certificates.

First clean any previous SSL development certificate from your machine. So, run the following command in your Command Prompt.

```
dotnet dev-certs https --clean
```



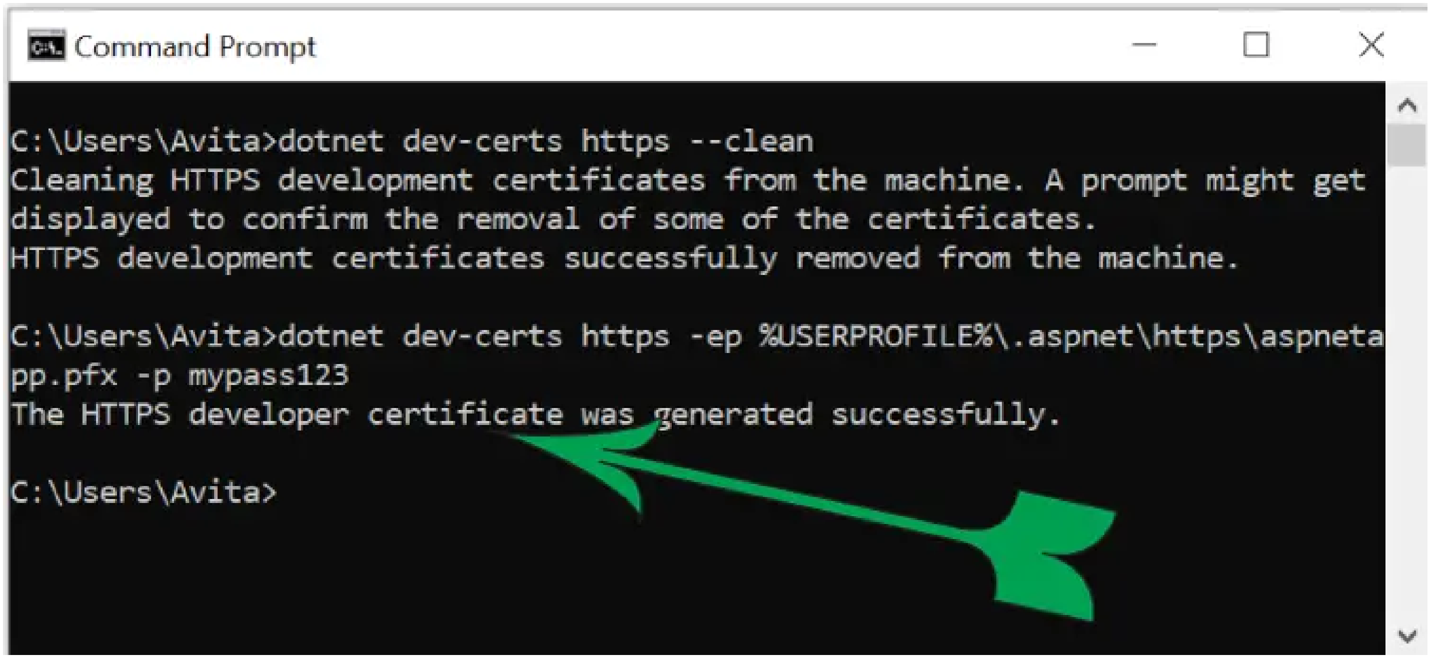
```
C:\Users\Avita>dotnet dev-certs https --clean
Cleaning HTTPS development certificates from the machine. A prompt might get
displayed to confirm the removal of some of the certificates.
HTTPS development certificates successfully removed from the machine.

C:\Users\Avita>_
```

Accept any prompt which you get. Then you will see a message – HTTPS development certificates successfully removed from the machine..

Next, run the following given command to generate a new **SSL Development Certificate**.

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
mypass123
```

You will receive a message – The HTTPS developer certificate was generated successfully.



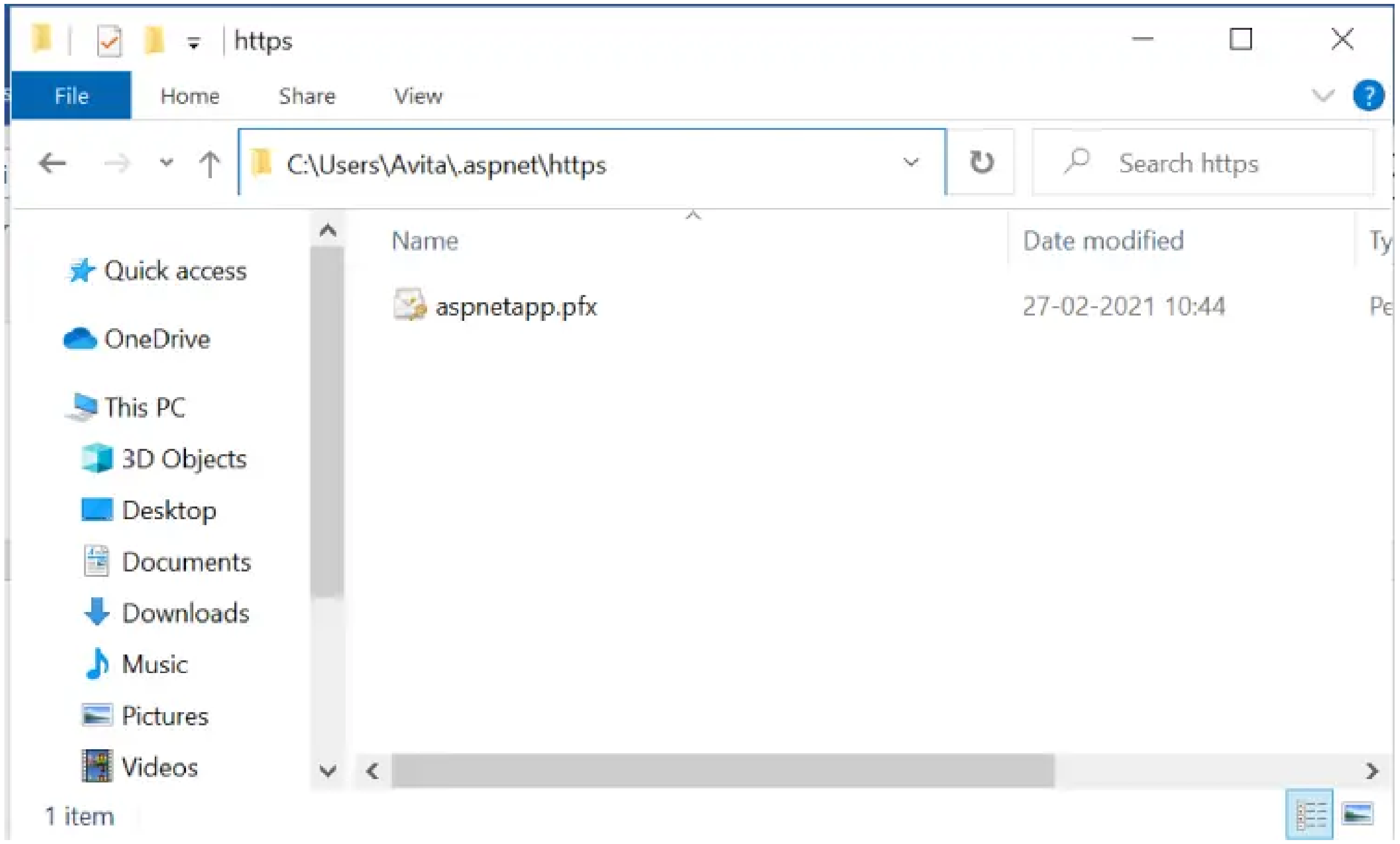
```
C:\Users\Avita>dotnet dev-certs https --clean
Cleaning HTTPS development certificates from the machine. A prompt might get
displayed to confirm the removal of some of the certificates.
HTTPS development certificates successfully removed from the machine.

C:\Users\Avita>dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspneta
pp.pfx -p mypass123
The HTTPS developer certificate was generated successfully.

C:\Users\Avita>
```

The SSL by the name of aspnetapp.pfx will be created with a password mypass123.

The path of the SSL certificate will be inside user profile folder, since I have referred it from %USERPROFILE%\. The full path of the SSL in my case is:

```
C:\Users\Avita\.aspnet\https
```

Here Avita is my windows login name, change it to your login name, and find it in your pc. In the below image I have shown the SSL certificate file which is generated on my pc.



Name	Date modified	Ty
aspnetapp.pfx	27-02-2021 10:44	Pe

The final thing to do is to trust the ASP.NET Core HTTPS development certificate. This command which does this is given below.

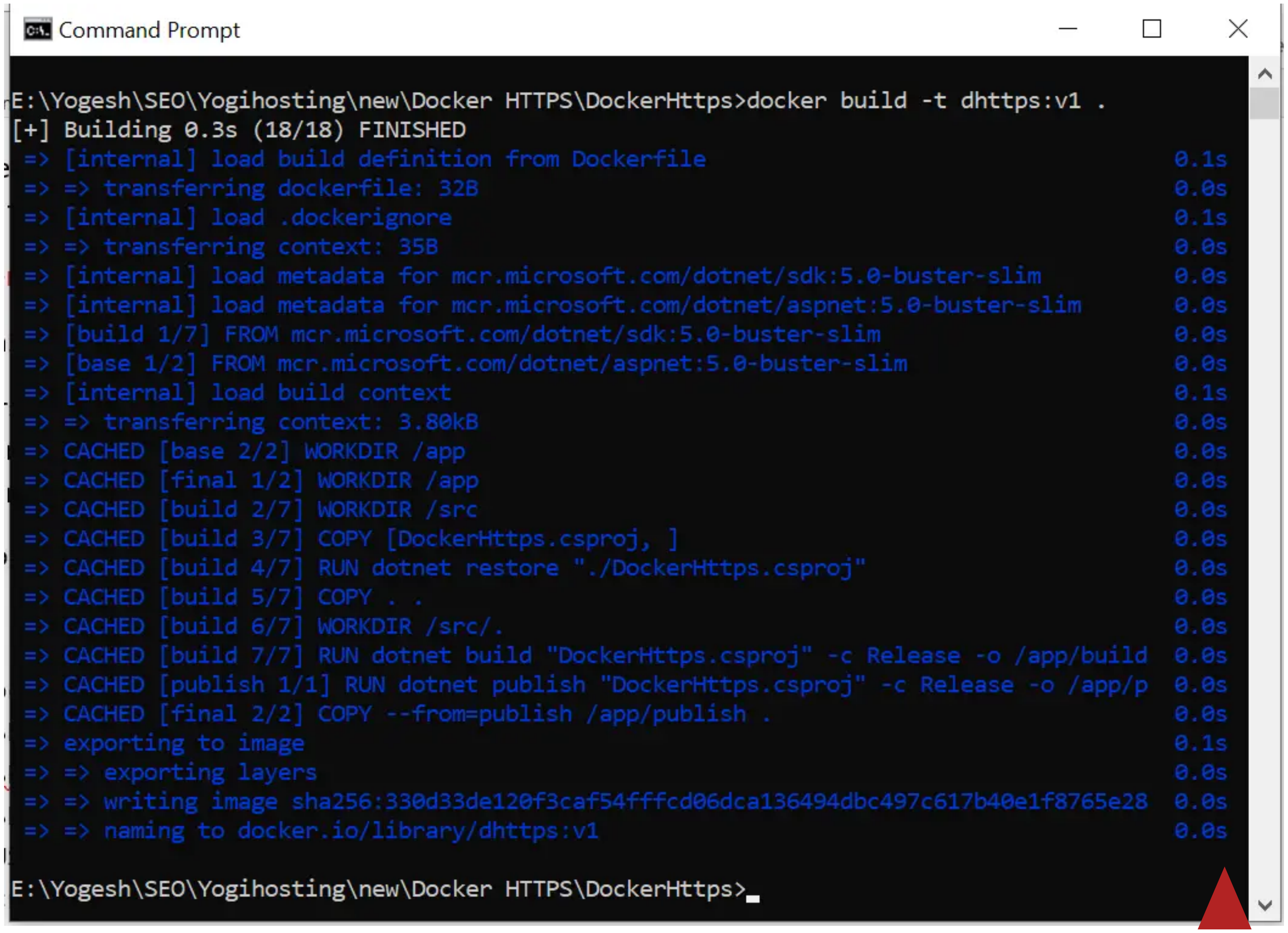
```
dotnet dev-certs https --trust
```

If you get a prompt after running the above command then make sure you accept it.

ASP.NET Core Docker Certificate in Volume

First, we need to build the Docker Image so that it contains our ASP.NET Core app. In your command prompt, go to the directory containing the Dockerfile and then run the following docker build command:

```
docker build -t dhttps:v1 .
```



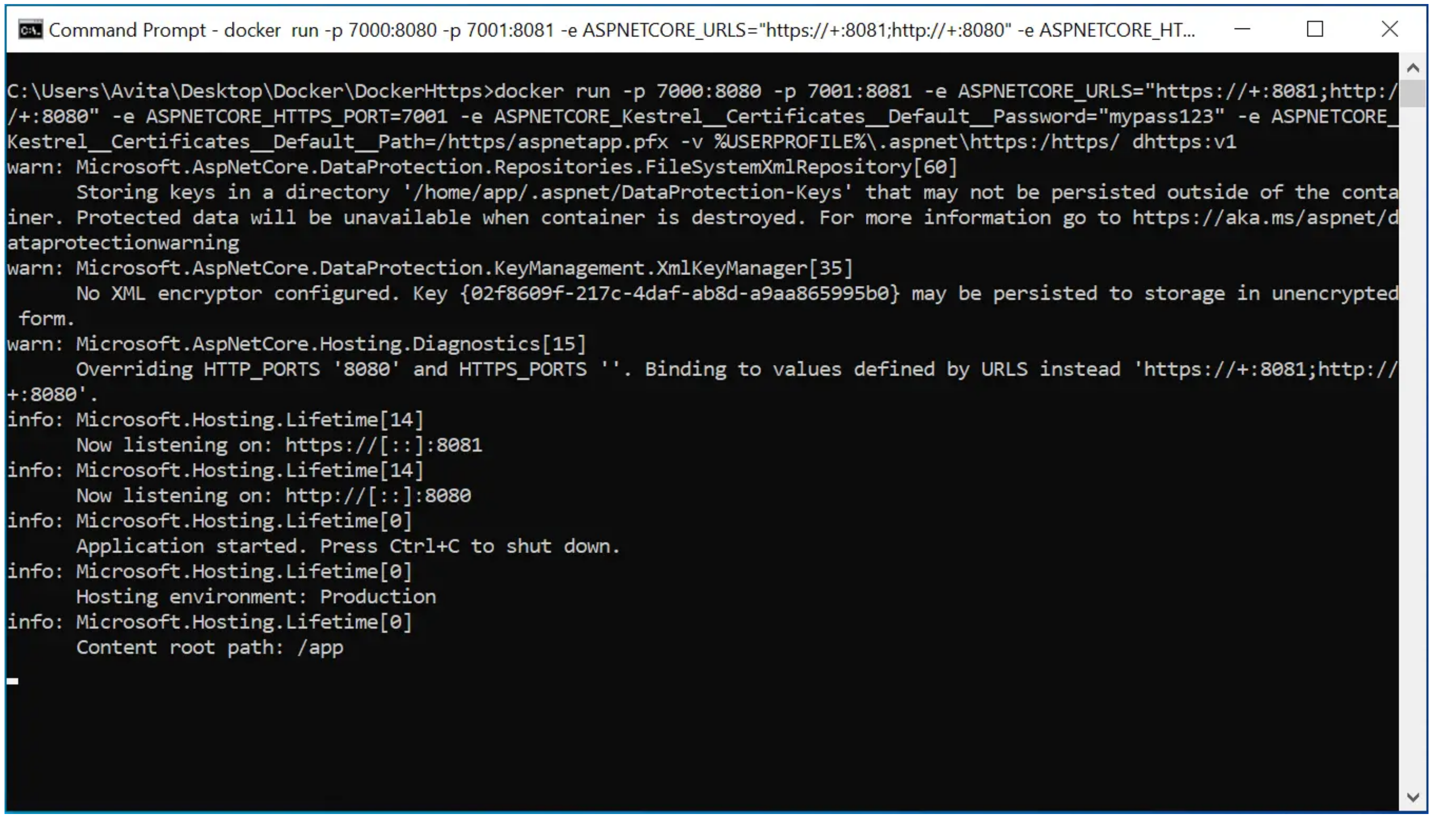
```
E:\Yogesh\SEO\Yogihosting\new\Docker HTTPS\DockerHttps>docker build -t dhttps:v1 .
[+] Building 0.3s (18/18) FINISHED
 => [internal] load build definition from Dockerfile                                       0.1s
 => => transferring dockerfile: 32B                                                        0.0s
 => [internal] load .dockerignore                                                          0.1s
 => => transferring context: 35B                                                           0.0s
 => [internal] load metadata for mcr.microsoft.com/dotnet/sdk:5.0-buster-slim              0.0s
 => [internal] load metadata for mcr.microsoft.com/dotnet/aspnet:5.0-buster-slim           0.0s
 => [build 1/7] FROM mcr.microsoft.com/dotnet/sdk:5.0-buster-slim                          0.0s
 => [base 1/2] FROM mcr.microsoft.com/dotnet/aspnet:5.0-buster-slim                        0.0s
 => [internal] load build context                                                          0.1s
 => => transferring context: 3.80kB                                                        0.0s
 => CACHED [base 2/2] WORKDIR /app                                                         0.0s
 => CACHED [final 1/2] WORKDIR /app                                                        0.0s
 => CACHED [build 2/7] WORKDIR /src                                                        0.0s
 => CACHED [build 3/7] COPY [DockerHttps.csproj, ]                                         0.0s
 => CACHED [build 4/7] RUN dotnet restore "./DockerHttps.csproj"                           0.0s
 => CACHED [build 5/7] COPY . .                                                            0.0s
 => CACHED [build 6/7] WORKDIR /src/.                                                      0.0s
 => CACHED [build 7/7] RUN dotnet build "DockerHttps.csproj" -c Release -o /app/build      0.0s
 => CACHED [publish 1/1] RUN dotnet publish "DockerHttps.csproj" -c Release -o /app/p      0.0s
 => CACHED [final 2/2] COPY --from=publish /app/publish .                                  0.0s
 => exporting to image                                                                     0.1s
 => => exporting layers                                                                    0.0s
 => => writing image sha256:330d33de120f3caf54fffcd06dca136494dbc497c617b40e1f8765e28      0.0s
 => => naming to docker.io/library/dhttps:v1                                               0.0s

E:\Yogesh\SEO\Yogihosting\new\Docker HTTPS\DockerHttps>
```

Docker image with the name of dhttps and tag v1 will be created for our ASP.NET Core app.

Next command is to run a docker container with the image we just built. This command is given below:

```
docker run -p 7000:8080 -p 7001:8081 -e
ASPNETCORE_URLS="https://+:8081;http://+:8080" -e ASPNETCORE_HTTPS_PORT=7001
-e ASPNETCORE_Kestrel__Certificates__Default__Password="mypass123" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx -v
%USERPROFILE%\.aspnet\https:/https/ dhttps:v1
```

See the below image which shows I am running this command.



Command Prompt - docker run -p 7000:8080 -p 7001:8081 -e ASPNETCORE_URLS="https://+:8081;http://+:8080" -e ASPNETCORE_HT...

```
C:\Users\Avita\Desktop\Docker\DockerHttps>docker run -p 7000:8080 -p 7001:8081 -e ASPNETCORE_URLS="https://+:8081;http:/
/+:8080" -e ASPNETCORE_HTTPS_PORT=7001 -e ASPNETCORE_Kestrel__Certificates__Default__Password="mypass123" -e ASPNETCORE_
Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx -v %USERPROFILE%\.aspnet\https:/https/ dhttps:v1
warn: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[60]
      Storing keys in a directory '/home/app/.aspnet/DataProtection-Keys' that may not be persisted outside of the conta
iner. Protected data will be unavailable when container is destroyed. For more information go to https://aka.ms/aspnet/d
ataprotectionwarning
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
      No XML encryptor configured. Key {02f8609f-217c-4daf-ab8d-a9aa865995b0} may be persisted to storage in unencrypted
 form.
warn: Microsoft.AspNetCore.Hosting.Diagnostics[15]
      Overriding HTTP_PORTS '8080' and HTTPS_PORTS ''. Binding to values defined by URLS instead 'https://+:8081;http://
+:8080'.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://[::]:8081
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://[::]:8080
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /app
```

Let us understand the above command:

=> With the "-p" option -p 7000:8080 -p 7001:8081 the ports 8080 and 8081 of the container are exposed to the ports 7000 and 7001 of the host.

=> I defined 3 environment variables to pass values to the app which is running inside the container. The environment variables are used to pass app's url, https certificate location and ssl certificate path. These are:

> The ASPNETCORE_URLS environment variable is used to specify the URL for the app like `ASPNETCORE_URLS="https://+:8081;http://+:8080"`. This means that

the APP will be opened in both http and https.
The ASPNETCORE_Kestrel__Certificates__Default__Password specifies the password for the SSL certificate – `ASPNETCORE_Kestrel__Certificates__Default__Password="mypass123"`. Recall it is mypass123.
The `ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx` specifies the default path of the https certificate. It is set to be inside the 'https' directory of the container. Note that from this path the certificate should load. Docker will copy the https certificate from a file location in our system which is outside the docker container to inside the https folder of the container. I will use **Docker Volume** to specify this path.

Next, with the volume "-v", I have specified where Docker should look for the SSL certificate on the drive. My certificate is outside of the container and is mapped to the container using a volume instead of bundling it together with the app in the image.

I used the below code for doing this work:

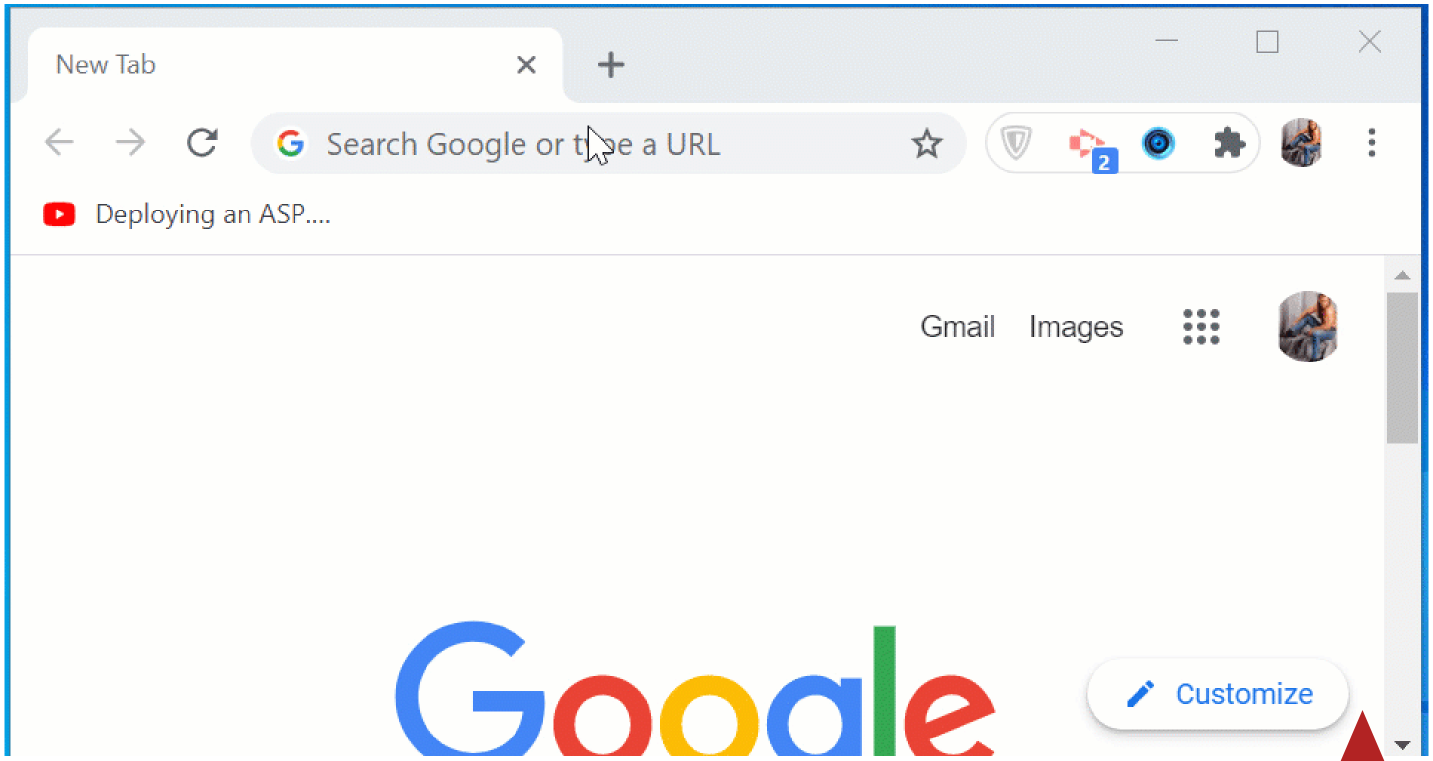
```
-v %USERPROFILE%\.aspnet\https:/https/
```

Finally, at the last of the docker run command, I specified the docker image to run inside this container – dhttps:v1.

★ I know this whole command is quite big to type on the command prompt. So I tell you a good news. You can keep all these settings like https ports, ssl, volumes, etc, in a "Docker Compose file" and simply use them from there. I have explained this on my tutorial called Docker Compose – Exposing ports and configuring Environment variables for HTTPS.

Testing

Now you can open the URL of the app in our browser, url is https://localhost:7001/. The app will open from the docker container with HTTP certificate fully working. I have shown this in the below video.



Check SSL Certificate inside the Container

We can now run a bash session and check the HTTPS Certificate inside the https folder of the container running our .NET APP.

First start the bash session with the following command.

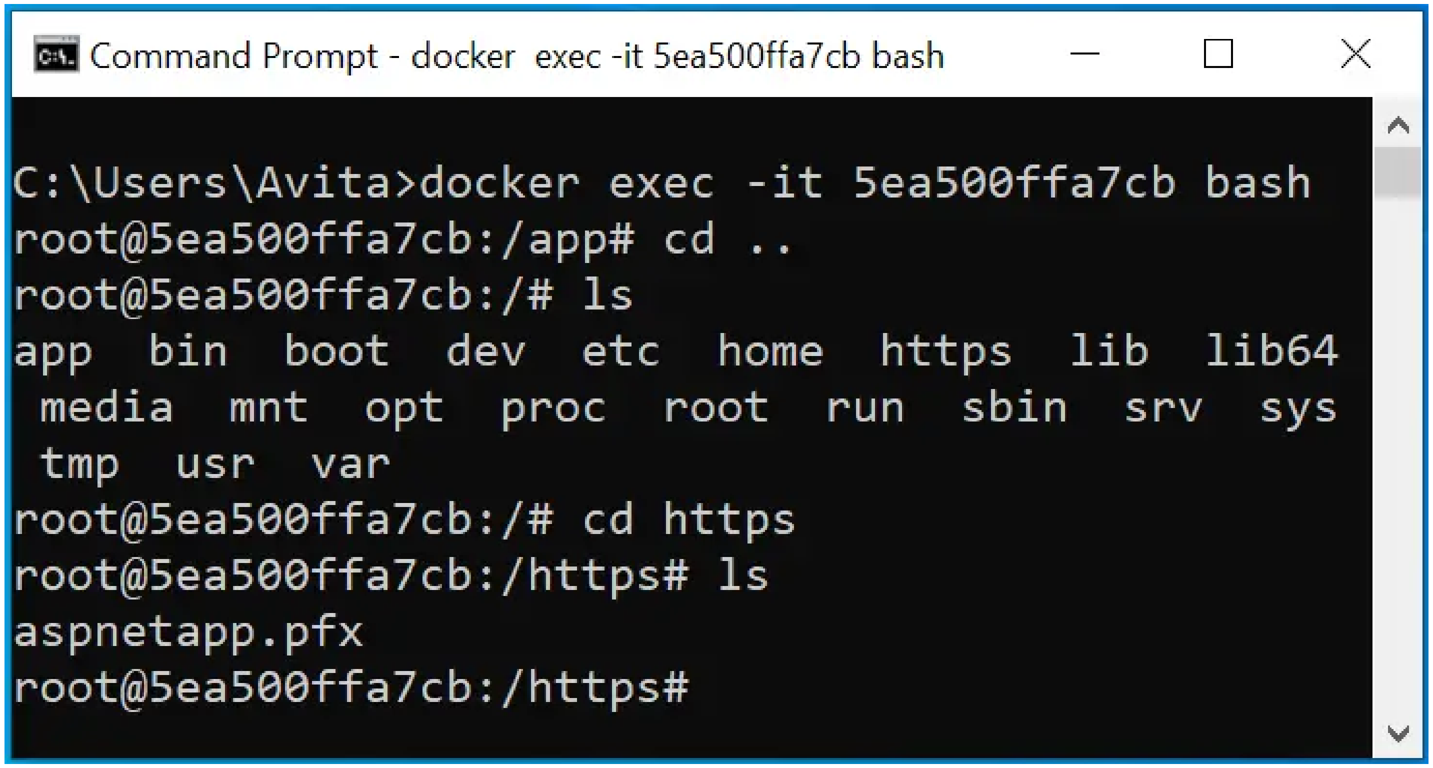
```
docker exec -it 5ea500ffa7cb bash
```

Here 5ea500ffa7cb is the container id.

Next run the `cd ..` command to go the root of the container. After that run the `ls` command to see all the directories. Here you will find the https directory which docker has created and copied the certificate from the volume.

Now go inside this directory from the command – `cd https` and then run the `ls` command. You will see the https certificate present there.

See the below image where we have shown this:



```
C:\Users\Avita>docker exec -it 5ea500ffa7cb bash
root@5ea500ffa7cb:/app# cd ..
root@5ea500ffa7cb:/# ls
app  bin  boot  dev  etc  home  https  lib  lib64
 media  mnt  opt  proc  root  run  sbin  srv  sys
 tmp  usr  var
root@5ea500ffa7cb:/# cd https
root@5ea500ffa7cb:/https# ls
aspnetapp.pfx
root@5ea500ffa7cb:/https#
```

Conclusion

In this tutorial you learn how to use HTTP certificate for ASP.NET Core app running in a Docker Container. You also learned how to generate development https certificate for ASP.NET Core app and the way to tell docker container about it's path by using volume mapping. Hope you liked reading and learning from it. Kindly share it on your facebook and twitter accounts so that other people can also learn this.

★ Next – Multi-Container ASP.NET Core App with Docker Compose