DOCKERS

Lab Book

Document Revision History

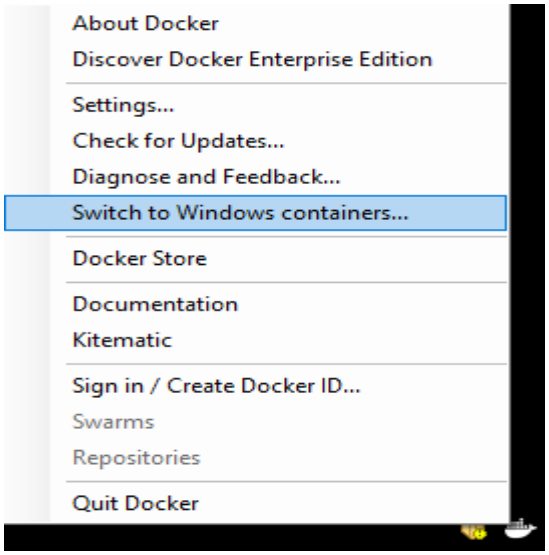
Date	Revision No.	Author	Summary of Changes
22nd Feb 2018	V1.0	Ashish Rajguru	Created the content for the same

Contents

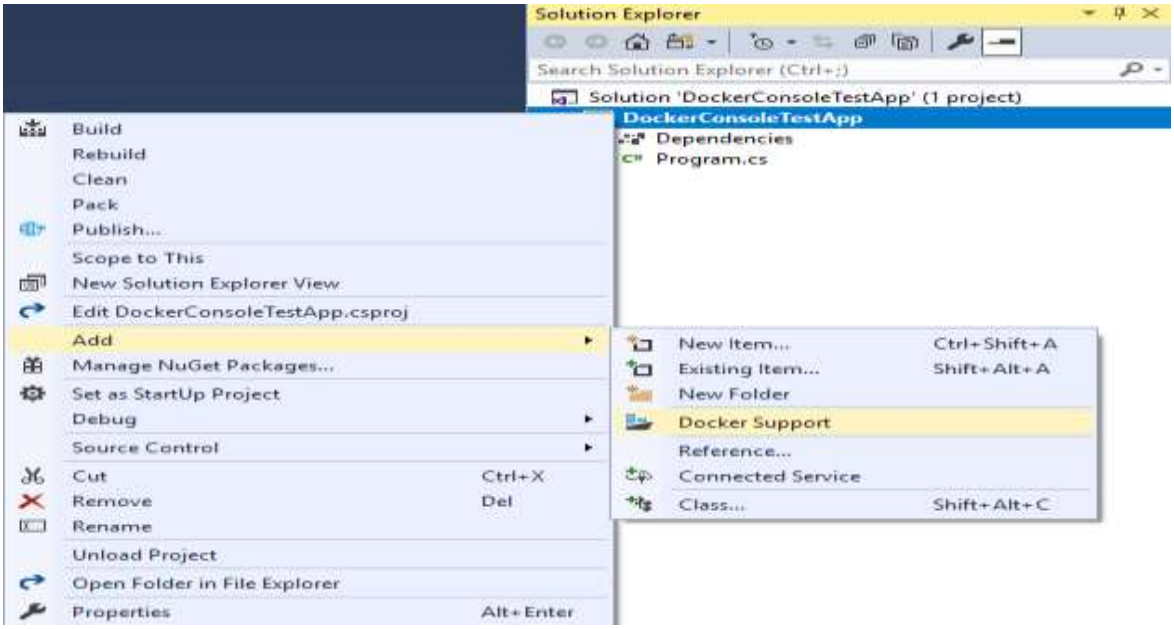
Lab1: To Containerize existing .NET application

Objective	This Lab will help you understand 1. How to Containerize .NET Console/Web Application
Time	60 Mins

1. **Configure Docker for Windows to use Windows Containers instead of Linux Containers**
   - Right-click the Docker taskbar icon in Windows and select Switch to Windows Containers, as shown in the figure below.



2. **Use Visual Studio to add Docker support to the application**
   - Right-click the project node and then select Add and Docker Support.
   - The Docker project template adds a new project to the solution called docker-compose. The project contains the Docker assets (simple .yml metadata files) that compose the Docker images and containers' startup settings you need, as shown in the figure below.

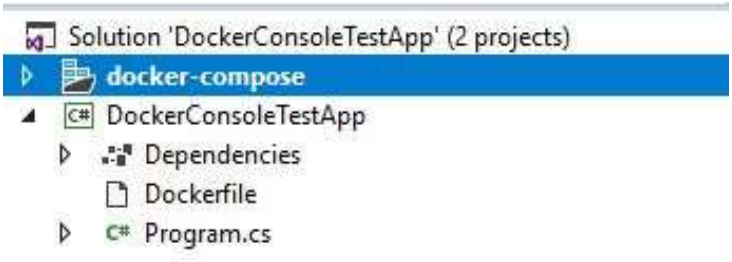


- The project and solution has been modified with the Docker metadata. The files added by the Docker project template are used to create the Docker image and launch a container.
- For each application or service in the solution Visual Studio adds a dockerfile to the project's root folder. Sample dockerfile added:

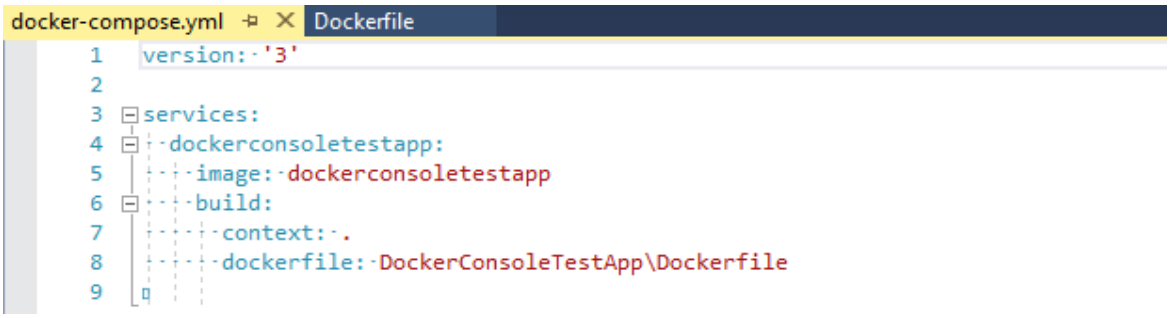
Dockerfile

```
1  FROM microsoft/dotnet:2.0-runtime-nanoserver-sac2016 AS base
2  WORKDIR /app
3
4  FROM microsoft/dotnet:2.0-sdk-nanoserver-sac2016 AS build
5  WORKDIR /src
6  COPY *.sln ./
7  COPY DockerConsoleTestApp/DockerConsoleTestApp.csproj DockerConsoleTestApp/
8  RUN dotnet restore
9  COPY . .
10 WORKDIR /src/DockerConsoleTestApp
11 RUN dotnet build -c Release -o /app
12
13 FROM build AS publish
14 RUN dotnet publish -c Release -o /app
15
16 FROM base AS final
17 WORKDIR /app
18 COPY --from=publish /app .
19 ENTRYPOINT ["dotnet", "DockerConsoleTestApp.dll"]
```

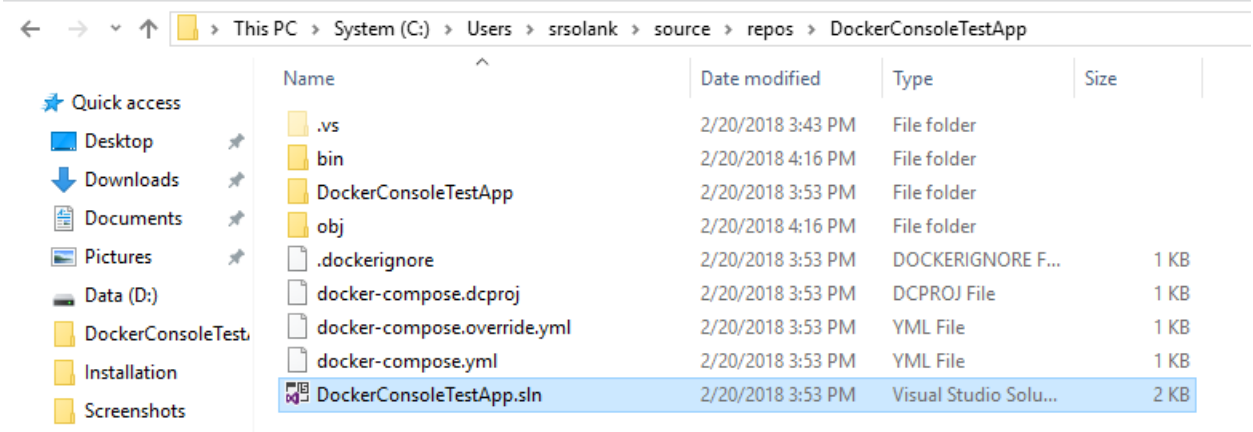

- The other files in the docker-compose project are the Docker assets needed to build and configure the containers. Visual Studio puts the various docker-compose.yml files under one node (project) to highlight how they are used.



- The base docker-compose file contains the directives that are common to all configurations/environments. The docker-compose.override.yml file contains environment variables and related overrides for a by-default developer configuration.



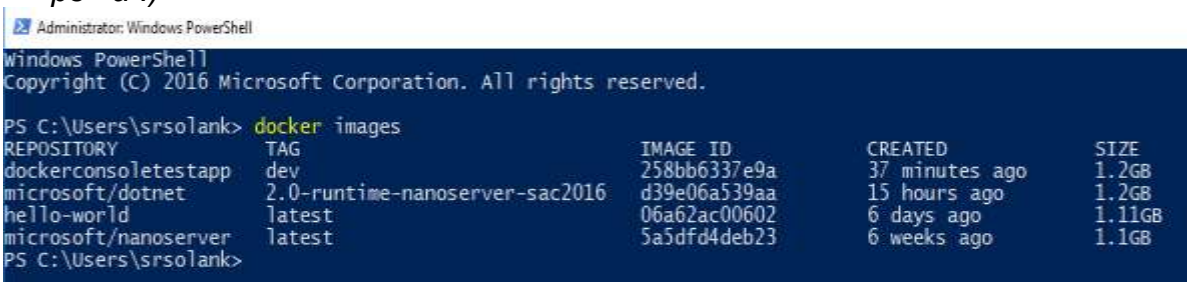
- The project structure after adding docker support:



- **Press Ctrl+F5 or F5 to compile the .NET application bits, create the Docker image and launch the Docker container all in a single step.** Visual Studio integration is part of adding Docker support to the solution, and just by running the

dockerized application with VS you are already creating the Docker images and deploying the containers in Docker, all in a single step.

(The installed images can be seen by running "docker images" using a command line tool in elevated mode. To get all the running container details execute "docker ps –a".)

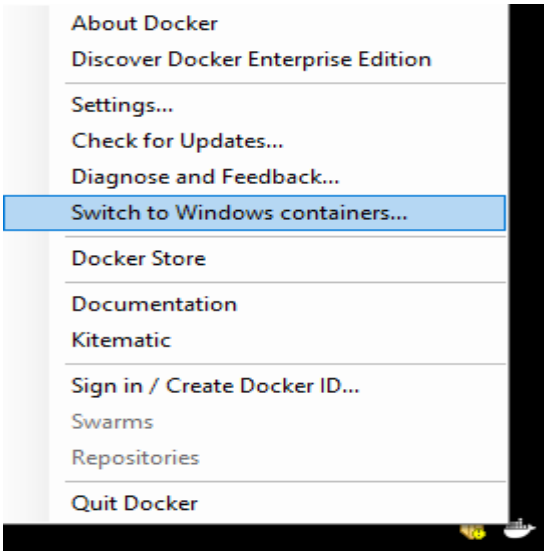


Lab 2:To Containerize existing .NET application with docker CLI

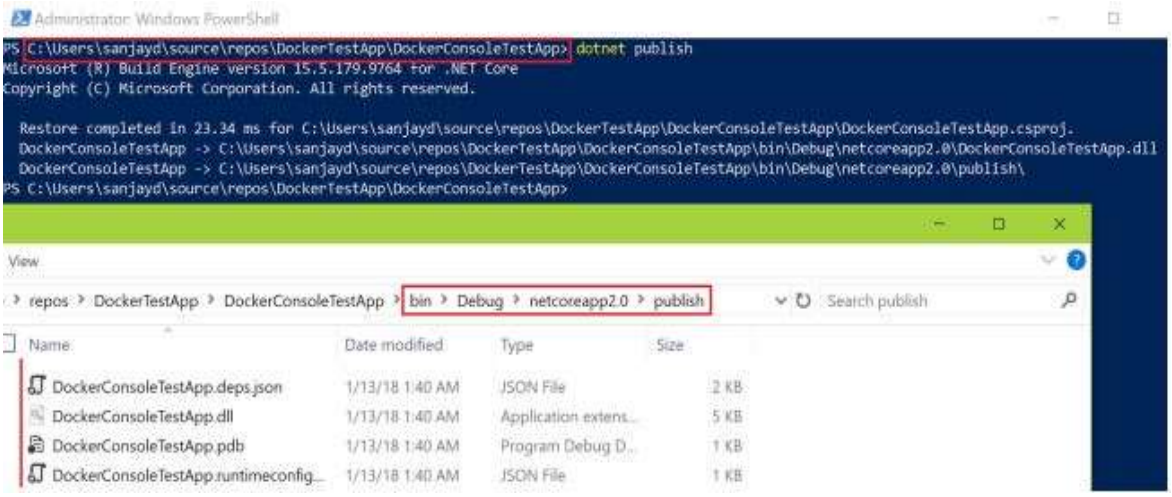
Objective	This Lab will help you understand 1.How to Containerize existing .NET application with docker CLI and manually adding dockerfile
Time	60 Mins

Containerize existing .NET applications with docker CLI and manually adding dockerfile:

1. **Configure Docker for Windows to use Windows Containers instead of Linux Containers**
   - Right-click the Docker taskbar icon in Windows and select Switch to Windows Containers, as shown in the figure below.



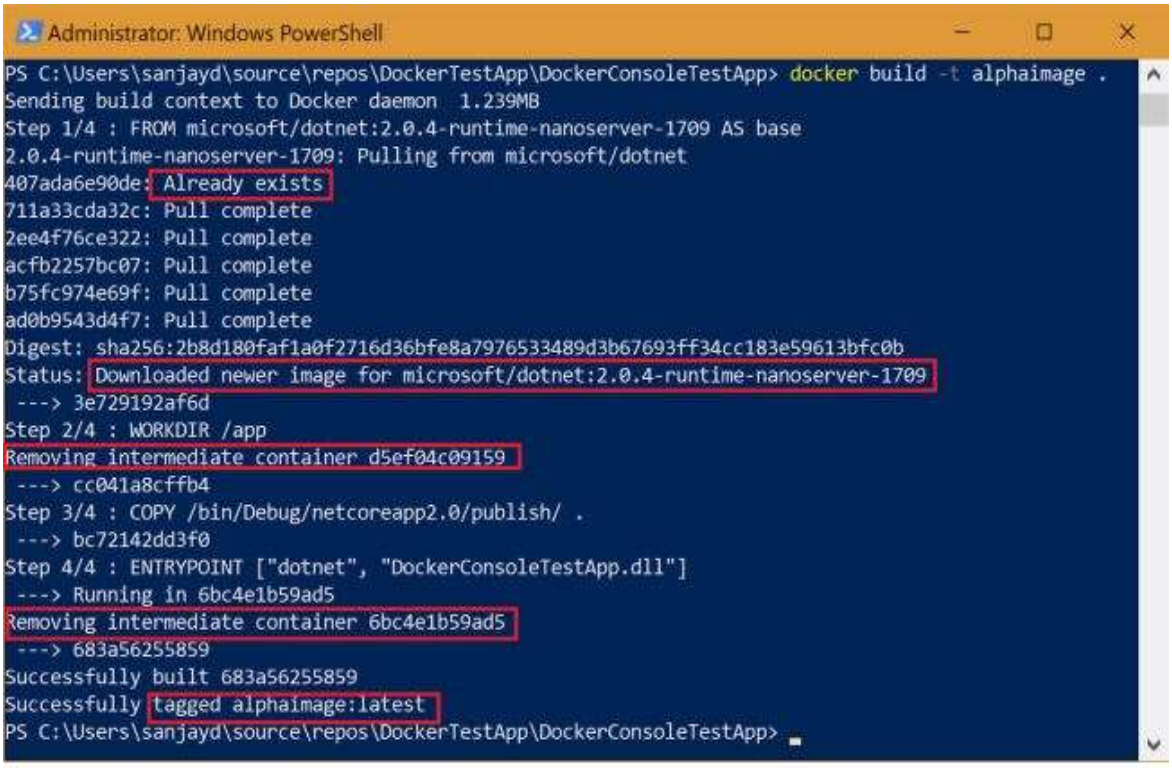
2. **Manually containerize the application**
   - Next publish the existing .NET console app. Run 'dotnet publish' command in PS/CMD (inside the project folder). Just to make sure all is good, you can fire up a PS/CMD prompt inside the publish folder and execute 'dotnet DockerConsoleTestApp.dll' to make sure the app is working fine.



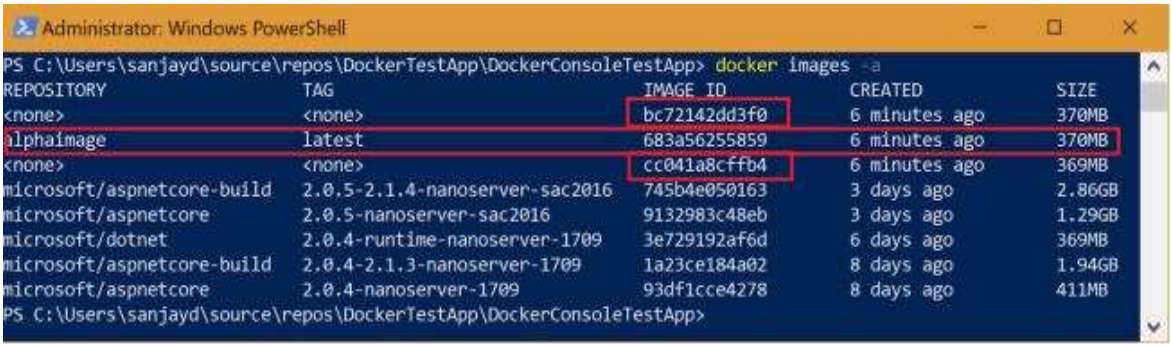
- Now create an image for the app using docker. Add a docker file in the solution and paste the content below. To read more about this file look here.

```
FROM microsoft/dotnet:2.0.4-runtime-nanoserver-1709 AS base

WORKDIR /app
COPY /bin/Debug/netcoreapp2.0/publish/ .

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

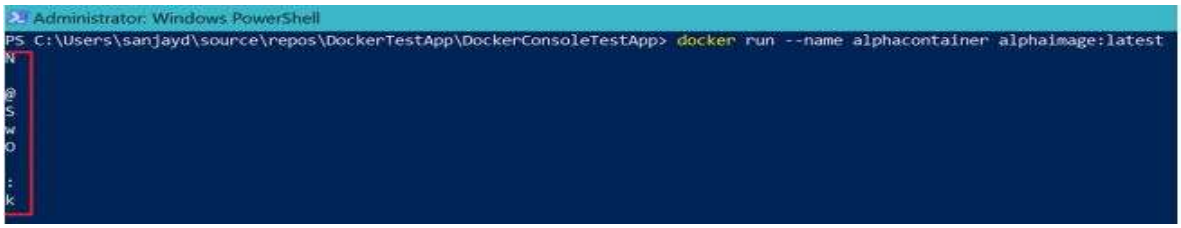
- Fire up a PS/CMD inside the project directory (where the dockerfile also resides) and run 'docker build -t alphaimage'. This tells docker to execute the docker file commands one by one and create/build an image named as 'alphaimage' (name has to be all lowercase). You can optionally add a tag too like 'docker build -t alphaimage:v1'. If nothing is provided as tag, 'latest' is considered. The docker engine creates the image. If the base dotnet image is not there in the local registry, docker will first download the same from docker hub. But later when you modify your code and run the same command again to build new image; docker will intelligently skip all downloaded images/sections.



- Run 'docker images -a' and check the output that gives all the installed images.



- The image contains the app we just built. Run it inside a container and validate. Execute the 'docker run --name alphacontainer alphaimage:latest' command from PS/CMD and see the app's output. (Press Ctrl+C to stop the execution.)



In the command we asked docker to create a container called 'alphacontainer' from the image called 'alphaimage' that has tag as 'latest'. The image already knows about the start-up path, so after creating the container docker automatically started the app. You can run the command again with a different container name, say "docker run --name alphacontainer2 alphaimage:latest". So now there are two containers running the same app using the development machine's OS as its base. To get all running container details execute 'docker ps -a' and you should see below. You can see both the containers are still running (the app). When we pressed Ctrl+C it only stopped to display the output in the PS window.