Demo Script: Building and Managing Images

Steve Lasker

[Steve.Lasker@Microsoft.com](mailto:Steve.Lasker@Microsoft.com)

<http://blogs.msdn.com/SteveLasker>

# Deck

# Demo environment setup & prereqs

# Demo Reset

**docker rm -f $(docker ps -a -q)**

**docker rmi alpine -f**

**docker rmi -f tradapp**

**docker rmi -f tradapp:optimized**

## Cache Images

**docker pull microsoft/aspnetcore:1.0.1**

**docker pull microsoft/aspnetcore-build:1.0.1**

# Demo Projects

# **Traditional Build**

You have an existing build system.

How many have .NET Full FX Web Apps?

How many have .NET Core Apps

Node?

Go?

We’re going to focus this talk on .NET Core using Linux to demonstrate where we’re heading with Windows as well. As Windows Containers become available, you’ll see the same experiences for Windows Server and full FX Apps as well.

Lets say you have an app based on .NET Core 1.0.0

You’ve got it deployed and being built with VSTS using a VSTS build agent.

The build agent, must have the dependencies to build your app. Such as .NET Core 1.0.0

# **Building .NET Core w/ASPNETCORE-BUILD Image**

|  |  |
| --- | --- |
| **Cd c:\VSTS\Trad\TradApp**  **docker run -it --rm -v "$pwd\:/sln" --workdir /sln microsoft/aspnetcore-build** | We’re starting here in our solution / root directory  We’ll start the ASP.NET Core Build container, passing in our current solution directory  We’ll make it the working directory |
| **# touch a.txt**  **# ls**  VS – Delete a.txt | We’re now running this container, with our solution volume mounted into the container.  Notice we can change a file in the container and see it back in VS.  From VS, we can delete the file, and it’s gone in the container |
| **Ls** | If we re-list the contents, we see it’s gone |
| **# dotnet restore** | Now that we’re in the container, we can run our normal commands  We can restore from the root of the solution |
| **dotnet test test/TradAppTests/project.json** | We can run our tests from the root, we just need to tell it the path to our project.json file |
| **dotnet publish src/TradApp/project.json -c release -o $(pwd)/publish** | Finally, we’ll create the published output, placing the contents in a bin folder. Also notice, we’re using the release configuration |

# **Building .NET Core *in the* ASPNETCORE-BUILD Image**

Using this model, we’ve just shown we can compile our app, test it, and publish the output in a container.

Because it’s in a container, we can run this anywhere.

We just need to automate this

|  |  |
| --- | --- |
| In the root of the solution: Add New Item  **build.sh** | We’re going to create a build script in the root of our solution |
| Copy the commands into the script file  **#!bin/bash**  **dotnet restore**  **dotnet test test/TradAppTests/project.json**  **rm -rf $(pwd)/publish**  **dotnet publish src/TradApp/project.json -c release -o $(pwd)/publish** | We’ll simply copy those same commands into the script file, with the shebang at the top |
| File 🡪 Advanced Save Options | Because we’re running this on Linux, we need to save with just LF endings |
| **docker run --rm -it -v "$pwd/:/sln" --workdir /sln microsoft/aspnetcore-build:1.0.1 sh ./build.sh** | We can now execute this script in the container |
| Add docker-compose-build.yml | Rather than typing the full commands each time, Docker has a nice way to capture all these parameters. |
| version: '2'  services:  tradapp-build:  image: microsoft/aspnetcore-build:1.0.1  volumes:  - .:/sln  working\_dir: /sln  entrypoint: sh ./build.sh | We can now put all these configuration options into the compose file  Notice the volume paths are a little different. We can use . as its relative to the location of the compose file.  There’s some other subtleties, like working\_dir instead of workdir |
| **docker-compose -f Docker-compose-build.yml up** | Now, we can simply call docker-compose, passing it our build definition, and say “up” |

# Building the release image

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| FROM microsoft/aspnetcore:1.0.1  WORKDIR /app  EXPOSE 80  COPY . .  ENTRYPOINT ["dotnet", "TradApp.dll"] | Now that we have the output ready to publish, we simply place that output into an ASP.NET Core Optimized image  We add EXPOSE to let docker run know what ports the container is “doing business on” |
| "publishOptions": {  "include": [  "Dockerfile", | The last thing we’ll do is make sure our dockerfile is copied to our published directory |
| **docker build publish -t tradapp:latest** | Build our final image |
| **docker run -d -p 8080:80 tradapp**  **Localhost:8080** | Test our final image |
| **Docker images** | Now we can see a pretty big difference between our :build and :latest image |

# **Copy, Restore, Run**

If you search the internet, you’ll see simplistic dockerfiles where everything appears to be contained in the same dockerfile. It copies the source, does a restore and calls docker run.

This model tends to work well for interpreted languages like Node.js.

However, for compiled languages like GoLang and .NET, we can get performance advantages for pre-compiling the code.

We can also get runtime size advantages of the image as we only need to deploy the runtime components.

But, seeing is believing

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| In the TraddApp, create **dockerfile.single** | We’ll add another dockerfile to our TradApp. We’ll name it .single for building and running in a single container |
| FROM microsoft/dotnet:1.0.0-preview2-sdk  WORKDIR /app  ENV ASPNETCORE\_URLS http://+:80  EXPOSE 80  COPY . .  RUN dotnet restore  ENTRYPOINT ["dotnet","run”] | We’ll defer back to our dotnet image, which has the sdk as well  We’ll simply copy the source and restore our packages  And a few other things like tell Kestrel to listen to port 80  Since we didn’t publish the app, we’ll just use dotnet run |
| cd C:\VSTS\Trad\TradApp\src\TradApp\  docker build . -f Dockerfile.single -t tradapp:single | We’ll switch into the TradApp source directory  And, build the dockerfile.single, tagging it :single |
| Docker run -it -p 80:80 tradapp:single | We can now run this image. |
| <http://localhost> | And, if we browse to the page, it seems to come up just as fast |
| DockerStartTimer.exe "-d -p 80:80 tradapp:single" "http://localhost" | But, lets not be fooled by perception. Lets measure the actual time  I have this little utility I hacked together to execute a docker command. The timer will measure the amount of time for docker run to execute, then how long it takes to start serving requests |
| docker rm -f $(docker ps -a -q) | We need to clear out the running containers |
| DockerStartTimer.exe "-d -p 80:80 tradapp:latest" "http://localhost" | Lets compare this to the optimized build |
| Docker images | And if the startup time wasn’t enough, lets look at the image sizes |
| FROM microsoft/aspnetcore-build:1.0.1  WORKDIR /src  COPY . .  RUN dotnet restore  RUN dotnet publish -c release -o /app  EXPOSE 80  ENTRYPOINT ["dotnet", "/app/src.dll"] |  |