Demo Script: ASP.NET Core Docker Multi Container

Steve Lasker

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

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

# Demo Reset

# Create New Project

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***From Windows*** | |
| **Create New ASP.NET Core Web Project**  **Solution: AspNetCoreMultiProject**  **Project: Web** |  |
| Add Docker Support |  |
| Change the Dockerfile:  FROM microsoft/aspnetcore:1.0.1 | Since we haven’t yet released this image, I’m going to change our FROM to reflect the image we’ll have released when your actually using the tools |
| web:  image: dockertools.azurecr.io/aspnetcoremultiproject/web${TAG} | We’re going to take this opportunity to fully qualify our image. The base URL represents the docker registry to place the image within. While the next node is an arbitrary namespace. While the last is the image name |
| F5 | Let’s see what this looks like while running in docker.  We’ve done nothing special, expect add the docker artifacts that help Visual Studio build the image |

# Debug and Release

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***From Windows*** | |
| ps> Docker images | If we look at the images, notice the image size of the image we just built when compared to the microsoft/aspnetcore image. Notice they’re the same size |
| **ps>docker ps** | To see what’s happing with our debug image, lets explore a few more details.  We’ll need the ID, so lets list the running containers |
| Docker inspect [id] eg: docker inspect 845 "HostConfig": {  "Binds": [  "/c/Demos/MultiProject/src/Web:/app:rw",  "/c/Users/stevelas/clrdbg:/clrdbg:ro",  "/c/Users/stevelas/.nuget/packages:/root/.nuget/packages:ro"  &&  "Mounts": [  {  "Source": "/c/Demos/MultiProject/src/Web",  "Destination": "/app",  "Mode": "rw",  "RW": true,  "Propagation": "rprivate"  },  {  "Source": "/c/Users/stevelas/clrdbg",  "Destination": "/clrdbg",  "Mode": "ro",  "RW": false,  "Propagation": "rprivate"  },  {  "Source": "/c/Users/stevelas/.nuget/packages",  "Destination": "/root/.nuget/packages",  "Mode": "ro",  "RW": false,  "Propagation": "rprivate"  } | If we type docker inspect with the first 2-3 characters of the image id, we can see the configuration of this running image.  Did you know you could use just the first few unique characters?  If we look closely, we can see the volume mapping |
| Open dockerfile | We see mostly standard information. However, you may not be familiar with BUILD ARGs. These provide for values to be optionally passed into a build command, from docker build, or docker-compose build. We’ll come back to this. But, notice our COPY command use $source which was defined above. |
| Open docker-compose.yml | If we look at the “production” compose file, we’ll see our standard declarations.  If we start with the build, the context and dockerfile are set. This provides docker-compose build the information to build the image based on these configurations. What’s cool is when docker-compose build is called, it will use the image name to tag the image for you. Later, when you call docker-compose up against the same compose file, it will just use the image name reference.  Looking at Ports, we see only one side of the port mapping defined. This tells docker to open port 80 on the container, but use a dynamic port on the host. This alleviates conflict of more than one container attempting to use the same host port |
| Open docker-compose.dev.debug.yml | Now, if we look at the debug version of the compose file, we see additional information.  Notice the BUILD ARG for source is being redefined to an empty directory. And if we drop down, we can see 3 volume mounts being added. Overriding our app directory, and adding the nuget packages and debugger.  This means each time we F5, we don’t need to rebuild the image, nor copy contents into the image. Now, we’ve experimented with image caching. It turns our that during development time, the fastest thing is to volume mount the contents and leverage the host machine to build the content. |
| **docker exec -it [id] bash** |  |
| Stop debugging Change to release mode  F5 again | To |
| **Docker info [id]** | Notice we only have the debugger volume mounted. |
| **docker exec -it [id] bash** |  |
| **CTRL P CTRL Q or exit** | To exit our bash session, we’ll use CTRL P then CTRL Q |

# Iterative Changes

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***From Windows*** | |
| Set to debug & F5 |  |
| Change a razor file | Add “razor file”  Notice the change is reflected. But how? |
| Open docker-compose.yml Point to:  - DOTNET\_USE\_POLLING\_FILE\_WATCHER=1 | Explain Razor file system watcher |
| Change a C# File | Explain why this doesn’t just work, … yet |

# .Add View for an API Call

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***From Windows*** | |
| Drag in:  **Controllers\Magic8BallController**  **Views\Magic8Ball**  **Views\Shared\\_Layout.cshtml** |  |
| F5 | With our View of our API Call in place, we can start our container |
| Click Magic | We can now see our Magic menu available. What will we get  Boom. Hmm. Lets inspect this a bit |
| Set breakpoint Magic8BallController:Index |  |

# Add API Image

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***From Windows*** | |
| Right Click References 🡪 Add Reference  Browse | What if we wanted to add a dll that someone else compiled?  We’d right click on the references and choose Add Reference  If we don’t have the project, we’d browse for the dll that we had locally.  What if we do the equivalent for Docker Images? |
| **ps> docker images** | Just as we’d add a binary reference to a dll, we can add a reference to a completed image  Let’s get the collection of images we already have to remember the fully qualified image name |
| Copy the api image name | We can simply copy the image name here |
| Docker-compose.yml Add:  **api:**  **image: dockertools.azurecr.io/aspnetcoremultiproject/api** | And we’ll paste it here in our compose file.  Now, this is a place where it would be really cool if we could leverage a language service to search our registries and auto-complete the images found… yeah, that’s on our backlog, but not yet… |

# Add API Project

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***From Windows*** | |
| ***Add*** New Project Name: **Api**  Location **/src** | We’re going to add a new project to this solution  Because I want it to sit side by side with our web project, I’ll place it under the src directory |
| Select Web API | We’ll chose the Web API template |
| Add Docker Support | We’ll enable this project with our docker assets |
| Update Dockerfile FROM microsoft/aspnetcore:1.0.1 | Same workaround here, just updating the image to the released image name for aspnetcore |
| image: dockertools.azurecr.io/multiproject/api${TAG} | Fully qualify the image name |
| Set the API project to Startup F5 | Lets first test out the API project under a container. We’ll set it as the startup project and hit F5  Notice our api/values controller shows up, with the results we’d expect to see |
| Stop debugging | Ok, now that we have the baseline, lets add our API code |
| Copy in: Models\EightBallAnswers.cs Controllers\Magic8BallController.cs | We’ll drag in our models and controller |
| Open Models\EightBallAnswers | We can see we put the standard answers in a class to use |
| Open Controllers\Magic8BallController.cs | And, here we simply do some random number to pull one of the answers out, or request it by index |
| F5 | Lets see this in action  Notice our values controller shows up again, we’re good to start |
| Browse to localhost:[PORT]/Magic8Ball | Notice we pulled the api namespace off our URL. We did this in our attribute. But, if we call the API, voila |
| Set the startup project [http://localhost:{ServicePort}/Magic8Ball](http://localhost:%7bServicePort%7d/Magic8Ball) | Wouldn’t it be nice if we could set the default startup project?  Here, where you’d expect it, we honor the startup project.  But, notice the {ServicePort} variable. This is where we’ll insert the dynamic port we get back from Docker. |
| Open the Compose file | Notice in the compose file, for ports, we used a special nomenclature. We created a host port, but with the dynamic syntax. By setting it to a single value, this tells the docker host to take the port we specify and allocate a dynamic available port on the host  The VS Docker tools then query docker for the host port for this specific running image |

# Test the API Project with the Web Image

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***From Windows*** | |
| ***API\docker-compose.yml*** web:  image: dockertools.azurecr.io/multiproject/web  ports:  - "80:80" | To reference the Web project, we’re going to add an image reference, just as we did prior, with the binary reference  Now, this is where it does get a little tricky. We’re going into a area of how you can, but it’s not that easy. We’re kina four wheeling here off road.  This is where our next round of investments are coming to make this easy, but let me show you how to stitch this together |
| Set the configuration to Release  Rebuild the Web Project | We have a choice of running the debug or release image of the Web container when testing our API container. The problem of running the Debug version is it doesn’t actually have any content. Remember, we volume mount the content into the debug versions. To have a viable image to run, we need to build the release version |
| F5 | With our image built, we can now debug our API project.  As we startup, the API project will use the “binary” reference to our Web Project.  We don’t yet have multi-project debugging. That’s coming |
|  |  |

# Demo environment setup & prereqs

## Install and Configure Docker For Windows Beta

[Setting Up Docker For Windows (D4W)](onenote:Piñata.one#Setting%20Up%20Docker%20For%20Windows%20(D4W)&section-id={3A79CD1D-D9A5-484C-85FE-7B734C48A4F4}&page-id={43D2985A-D59C-455D-A413-69A585BBC0C8}&end&base-path=https://microsoft.sharepoint.com/teams/CPT/AzureTools/Shared%20Documents/Docker/Docker%20Investiga)

# Cached Docker Images for ASP.NET

Run the following in a PowerShell prompt

# Demo Reset

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

## Cache Images

**docker pull microsoft/aspnetcore:1.0.1**

# Reset All Images

Only necessary to clear all, and start from scratch.

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

**docker rmo -f $(docker images -q)**