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 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**

# Create New Project

|  |  |  |
| --- | --- | --- |
|  | **Demo step** | **Talk track & notes** |
|  | ***From Windows*** | |
|  | Create New ASP.NET Core Web Project  **Solution: Multi**  **Project: Web**  **w/Docker Support** (Checkbox) | We’ll create a new solution, and we’ll separate the solution name from the project.  First, we’ll add the web project |
|  |  | Notice we have a new web project, with the docker files already added |
|  | Add New Project ASP.Net Core Web  Name: **Api** | Now, lets add a new Api project, again with Docker enabled |
|  | Open Dockerfile | At this point we have two individual projects with docker support. Each has a dockerfile for its image definition.  With Visual Studio, you have a csproj file to define the definition of your project, and how it should build.  A dockerfile is the environment definition. I defines the OS, the components in the OS, our app, and any other environment information |
|  | Open docker-compose.yml | Each project has a docker-compose.yml file. This is where we can define the collection of services we want to instance.  You might think we need just one, the one we’re working on here. But, what if we wanted to spin up a sql container, a redis cache. Or, a service another team built for us. |

# Debug Multiple Containers

|  |  |  |
| --- | --- | --- |
|  |  | Since we’re going to build a 2-service solution, and we want to debug both of these, we’re going to add solution level docker support. |
|  | Web Project 🡪 Add 🡪 Docker Solution Support | Adding Docker Solution Support will place our docker-compose files in the solution. We didn’t move them, as you might have unique configurations when running just the single project. |
|  |  | We also didn’t use the VS Multi project debugging as we’re not actually starting up each project. To remain true to docker, we want to “compose up” our containers, with the service discovery docker provides. We may also want to instance some additional containers.  We considered deployment projects, but they also didn’t feel consistent with the docker model. It was important we met the tools with the technology, not morph the technology to meet our tools. |
|  | Open Solution\docker-compose.yml | Looking at our solution level compose, we can see it references our web project |
|  | API Project 🡪 Add 🡪 Docker Solution Support  [Yes to All] | We’ll do the same thing for our Api project. In this case, we merge the content into the solution. |
|  |  | We can now see our two services.  Docker will build these when it doesn’t find the images.  VS will call docker-compose up with the -build flag when the files are “dirty”  We could also add additional images, that might have been built by other teams. Perhaps an Auth service, a caching service, … I don’t need to load all the code here.  But, that’s getting a bit ahead. Let’s just run these two apps. |
|  | Click **Docker: Debug Solution** | Now that we have our solution configured, let’s start up our solution level compose. |
| Output window shows builds |  | We can see all the docker commands running here in the output window. VS is compiling the code and building an image based on our project dockerfiles.  We then use volume mounted drives from the container to our solution directory. This allows us to edit the files directly from VS. |
| C:\Users\MSSPEA~1\AppData\Local\Temp\SNAGHTMLf37774.PNG | Click the tabs of the two browser pages | As VS launches our app, we can see two browser pages open.  One for our Web App, one for our API.  When these are created, we don’t actually know what you want, so we open them both, allowing you to test them. |
|  | Open **Api\Controllers\ValuesController.cs**  Set a breakpoint | If we set a breakpoint in the values controller and refresh the page |
|  | Refresh the page | Notice our breakpoint hits, just as we’d expect it to |
| C:\Users\MSSPEA~1\AppData\Local\Temp\SNAGHTMLfc26d7.PNG | Stop Debugging | We’ll stop this debugging session and make this more interesting |

# Edit & Continue

|  |  |  |
| --- | --- | --- |
|  | Double click- API Project Properties | First, we’ll get rid of the 2nd API page opening. |
|  | Blank out the Launch URL value | For RC, we need to blank out the launchUrl value. We missed the launchBrowser flag, but we’ll get that fixed. |
|  | Drag the contents of DemoAssets\Api to the Api Project | We have a few files here for predicting the future. |
|  | Drag the contents of **DemoAssets\Web\Controllers** to the Web Project | We’ll add the right files to the web project as well. |
|  | Docker: Debug Solution | We’ll start up the solution level debugger and give it a try |
|  | Click Magic | As we click the magic link, we’ll start things up |
|  | Point to the missing image | The first thing you might notice is we have a missing image. |
|  | Open Folder on **Web\wwwroot\Images** | We could drag the files into the project, but I want to show off a little thing that can mean a lot. |
|  | Drag **Demo\Assets\Web\Images** to **Web\wwwroot\Images** | Lets copy the images to our project directory. Notice we’re doing this in Explorer. We don’t have to drag them into VS so they get added to the .csproj file. The VS 2017 csproj file brought the best parts of xproj and you no longer have to manage merge conflicts. It just works like any other code folder.  It’s the little things that can have such a big impact on mutli-team productivity. |
|  | Refresh the browser page | Without stopping and restarting, or rebuilding our docker image, we can refresh the page to see our contents. |
|  |  | The more you learn about containers, the more you can appreciate how much VS just made things simple for you… |

# Debugging Across Services

|  |  |  |
| --- | --- | --- |
|  |  | Let’s do a bit more complex… How would we debug across both containers? |
|  | Web\Controllers\Magic8BallController.cs | We can start with our website, where the action starts.  WE can set a breakpoint in the View Controller *in the website* |
|  | Api\Controllers\Magic8BallApiController | And another in the Api controller |
|  | Refresh the page | If we refresh the page, we’ll reload our View controller |
|  |  | We see our Web controller get hit  We can hover our variables, as you’d expect from VS |
|  | Continue |  |
|  |  | And our Api Controller. |

# Summary

Using Visual Studio 2017 RC with the Visual Studio Tools for Docker, we’ve just debugged individual projects and multiple projects. We’ve shown how dockerfiles in the project defined each project image. We’ve shown how solution level compose files can define a multi-container debugging session leveraging the discovery services offered by Docker.

You’re now ready to configure continuous delivery to deploy your mutl-container workload to Azure Container Services