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)**

**docker rmi -f $(docker images --quiet --filter "dangling=true")**

**az login**

**az acr login -n stevelasuap**

## Cache Images

**docker pull microsoft/aspnetcore:1.0.1**

# Dockerize a project

|  |  |  |
| --- | --- | --- |
|  | **Demo step** | **Talk track & notes** |
|  | Open existing project | We’ll us an existing .NET Core project that has existing project references |
|  | **F5** | Of course we can debug this under IIS Express Locally. But it’s a good reference point. |
|  | Click About |  |
|  | Open Controllers\HomeController.cs  Set breakpoint in About()  Refresh the page | If we make a change here, we can immediately see it |
|  | Stop debugging | But how do we enable docker here? What will my dockerfile look like?  Since .NET is a compiled language, how do I incorporate compilation, publishing to create an optimized image? |
|  | **Add Docker Support**  Choose Linux | We can easily add the default docker artifacts, including:  **Dockerfile** for building our image: that latest even uses multi-stage dockerfiles that fully encapsulates the build process, so there’s no differences in your builds when you run them locally or when you run them in your build system. |
|  |  | Notice the build section:  Notice VS was able to determine the project references and copied the projects for us, doing a restore.  This is where we take advantage of some performance best practices. By copying the project files in first, and doing a dotnet restore, docker will cache these layers. Until you make a project or nuget addition, subsequent builds will benefit from previous nuget restores and docker image caching. |
|  | Open the compose file | **Docker-compose.yml** file for defining the runtime instance information. Including the image names. We’ll see this a bit more later. |
|  | **F5** | We’ll now start debugging, this time under a docker container |
|  | Click About | Notice our breakpoint in the HomeController still works, just as you’d expect. But realize, we’re no longer debugging on my laptop, we’re debugging in the container |
|  | Open Views\Home\About.cshtml  Change  Refresh the browser | If we edit the About razor page, we can still make real time changes.  We’re viewing the app running in a container, but we’re still able to make changes with VS. Pretty cool, no? |
|  |  | This is all dune under docker, using standard docker commands. There’s no magic. Ok, anything can be magic if you don’t know what it is |
|  | Solution Add Existing Item  Open ***obj\docker\docker-compose.vs.debug.yml*** | If we add the auto-generated VS debugging file, we can see where the additional commands are added. |
|  | Open …vs.debug.yml | Here we can see al lthe volume mounts we’re using for source code changes, enabling realtime edits  NUGET Caches and the Razor file watcher to dynamically recompile our pages once changed.  These are auto-generated as these change as docker and our tools evolve. |

# Demo 2 – Commandline

|  |  |  |
| --- | --- | --- |
|  | Open PowerShell  CD to the Solution directory | Now that we have multi-stage dockerfiles, it’s easy to build our images, completely from a commandline. |
|  | **docker build . -f .\Web\Dockerfile**  **docker images** | We can issue a docker build command to trigger our multi-stage build  There is one caveat. Visual Studio is designed around solutions. Meaning, we likely have multiple projects. Each project is typically placed in the root of the solution.  As is with this project, the Website has dependencies on additional projects. Not API Services, but shared dlls.  We need to get the data and models dll into the same Web container to be built.  So, the visual studio scaffolded docker assets assume a solution level context. However, we also wanted to keep the dockerfiles within each project.  Which does mean we need to pass in the location to a docker build  If we look at our list of images, we’ll notice our image wasn’t named. We could pass a -t parameter and name it, but that gets frustrating to keep remembering all the parameters |
|  | **Docker-compose build** | As it turns out, VS scaffolded all this configuration information, including the image name in the compose file  So we can simply call **docker-compose build** |

# Demo 3 – Modernize with Additional Services

|  |  |  |
| --- | --- | --- |
|  | Add a new ASP.NET Core Web Api project  Project Name: **API**  Location: **Default** (under the solution) | We’ll add a new .NET Core Web API App |
|  | Choose Web API | WE’ll chose the Web API template as we don’t need all the pixel painting stuff, just the services  We can even enable Docker Support right here in the Create New dialog |
|  | View DockerFile | We can see the same multi-stage dockerfile here, slightly different for the project specifics, such as the donet run command |
|  | Open docker-compose.yml | If we look at the docker-compose file, we can now see two services are listed.  As you add additional containers, VS will add them to the compose file as we assume you want to debug these. Of course, you can tweak them as VS simply scaffolds the defaults. This is your code.  You might ask why a separate container:  In the VM world, you’ll likely put both the Web and API on the same VM. Or, wherever you place them, you need to manage the deployment of code and configuration.  IN the container world, we crate separate images for each process. Generally speaking a process = a container. So, each is its own image.  But, we do want to instance them together. |
|  | Drag in:  \_demoAssets\Api  \_demoAssets\Web | To make this a bit more interesting, lets include some additions  We’ll add a magic 8 ball to answer some questions from an API controller. |
|  | Open Web\Controllers\Magic8BallController  Breakpoint in Index() | Here we can see we’ll make a HTTP Web Request of our API controller  If we look above, we can see the URL is simply API. This comes from our compose file where we defined the service. The service name is the DNS endpoint our orchestrator will configure for us.  We can set a breakpoint here to show the flow. |
|  | Open Api\Controllers\ Magic8BallApiController.cs  Breakpoint at Get() | Likewise, we can open the Api controller and set a breakpoint here in the Get() method. |
|  | F5 | We can now restart our debugging session.  This time VS will not only compile and hook the debugger on the Web container, but it will hook the Api container as well. It will spin up any image listed in the compose file.  Let’s say you had an image built by another team, or an image that has RabbitMQ or HAProxy. VS knows which to hook the debugger to, and which not based on the builds. |
|  | Click Magic | As we click into the Magic view, we can see our flow kick in  First we hit the View controller.  We can see all the normal debugging information through hovering we’ve come to expect |
|  | F5 | If we continue debugging, we now step into the API container.  This is a whole ‘nother container, also running Linux, on my windows machine. |

# Demo 4 - Unit Tests

One of the key benefits of Docker is the ability to automate the building of our containers and the deployment. Deployment may occur as the result of a code check-in or a base image update for an OS or Framework patching event.

|  |  |  |
| --- | --- | --- |
|  | Add New Project  .NET Core  xUnit Test Project (.NET Core)  Name: **Web.test** | One of the first steps in automation is toenabling automated tests.  Let’s add some unit tests to our solution |
|  | Assert.Equal(1,2) | Lets add a deep test  We did say first steps. Don’t we want to know our first steps were good? In this case, good means we know how to fail our test to know the tests “worked” |
|  | Open dockerfile  FROM builder as test  WORKDIR /crc/Web.test  RUN dotnet test | Now that we have our unit tests, we’ll want to enable these in our multi-stage dockerfile  In between the build and publish, we’ll take the compiled output and run our test. |
|  | Open PowerShell in the sln directory | We can now run docker-compose build again, this time our unit tests will run |
|  | View the failure | Notice our failure |
|  | Change;  Assert. Equal(1,1) | Like a good developer, we’ll fix the error, …by fixing the test. |
|  | Docker-compose build | If we run docker build again, we’ll see our tests are now working. Surprise…😊 |

# Demo 5 – Azure Container Registry

|  |  |  |
| --- | --- | --- |
| Portal | Aka.ms/publicPortal  Aka.ms/acr/portal/preview | Whatever cloud you work in, you’re going to be pushing and pulling a lot of images. Images can get large. Even if they’re small, do you want to waste time pulling images across the network from far away?  While layer caching makes incremental pulls more efficient, you’ll likely be pulling complete images to new nodes or hosts.  So, another best practice is creating a registry network-close to your deployment. In your cloud, you’ll want the registry in the same region, not just the same cloud as your deployment.  But, what happens when you have multiple regions? Hmmm  If we open the Azure portal, we can create a new Azure Container Registry |
|  | Select Container Registry | We’ll start by creating an Azure Container Registry |
|  | Create New  Defaults w/standard  Switch Admin on | We’ll create a new registry  We have a few different SKUs here  Classic as it sounds, is our original option. Within the next week, we’ll be releasing the new managed Registry SKUs with general availability, access from all public Azure Data Centers, with full support. |
|  | Wait for completion | Here we have our newly created registry  Now we can do the stanard docker push/pull and see information about our registry  For each SKU we have different size constraints and you can easily see your current usage here  You can also configure vulnerability scanning solutions through our partners like Aqua and Twistlock |
|  | Click Geo Replication | How many of you run multiple data centers?  They may be multiples within your On Prem deployments  Or, multiple Azure Data Centers  The reason most run multiple data centers is two fold:   * Having a live backup * Being able to serve customers from a local presence   To support multiple regions, do you want different code bases?  Or, do you have the same codebase, but may provide localized content? Content in different languages, local product catalogs and pricing  A best practice for containers is to run a private registry in the same datacenter as your deployments. Containers are pulled often. New nodes come and go. You’ll be pulling lots of layers where the network traffic does add up.  Pulling across data centers adds latency, costs as you’re paying data center egress, and unreliability. It just happens.  With Azure Container registry, you can now configure a single registry to be replicated across multiple data centers  You manage a single registry, and each push/pull operations will be routed to the closest |
|  | Add Additional Geos | To replicate, we simply click on the map for the regions we want replicated  But, we see that geo replication is a feature of the premium SKU  With premium registries, we improve performance and throughput by synching multiple storage accounts to enable concurrent downloads |
|  | Switch to Overview | We can easily switch to Premium registries by switching to the overview page and updating the configuration |
|  | Switch back to Geo Replication | We can now add additional regions  Just click the additional regions and save  Depending on how big your registry is, your additional regions will be available once the sync is complete. However, ACR will add the Traffic Manager configuration, once complete. So you can continue to pulls, and once the sync is complete, the pulls will be local |
|  |  |  |

# Demo 6 – Geo Replication

|  |  |  |
| --- | --- | --- |
|  | Open two geo replicated sites  <http://stevelascentralusweb.azurewebsites.net/>  <http://stevelaseastusweb.azurewebsites.net/> | Here I have two different deployments.  For the most part, these are exactly the same.  I’ve added one dynamic element on the about page  Using the same registry URL, where is it being served from?  We have two deployments in two azure data centers.  Traffic Manager will route requests to the closest point. In this case, you never leave the data center. |
|  | Open About.cshtml  Change text | We’ll make a change here to our About page, just to prove we have a change. |
|  | Docker-compose build | We’ll rebuild our images with the latest content |
|  | Open Web\Controllers\HomeController.cs  Open About() | While our build is happening, lets look at the dynamic content that will be displayed based on the regional deploynent |
|  | Open docker-compose.yml  Add  ${DOCKER\_REGISTRY}demos/web:1 | Lets tag our image, which we can do in our compose file as well as the commandline |
|  | Docker-compose build | We can rebuild our images with the updated tags, run our tests, and have complete images |
|  | Docker-compose push | We can also leverage the compose push API to push the collection of images we’re working with |
|  | Open Portal – Webhooks | So, how does this work?  When you push to ACR, the push will go to the closes registry. There is no master. Push/Pull will always route to the closest node. ACR will then replicate to the other replicas |
|  |  | You can configure regionalized webhooks to fire when an image arrives at a replica  Based on these webhooks, we can trigger tests and/or deployments |
|  | Open Browsers | At this point, we should be able to see our updated deployments |
|  | Click to about | We can now see the different IP addresses |

# Summary