Demo Script: Docker – from the earth to the universe

A walkthrough to help developers learn the various docker primitives, and get the broader context, to scalling in container orchestration systems.

Steve Lasker

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

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

# Deck

<https://github.com/SteveLasker/Presentations/tree/master/DotNetConf2016>

# Demo environment setup & prereqs

## Creating Docker Hosts in Azure

docker-machine create -d azure --azure-subscription-id "[your\_azure\_sub\_id]" --azure-open-port 80 mydockerhost

# Reboot Tasks

If using VirtualBox

Once you reboot, you’ll need to do the following actions:

**docker-machine start default**

**docker-machine env default | Invoke-Expression**

If a network change has happened; moving from LAN to WiFi

**docker-machine restart default**

# Demo Reset

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

**docker rmi -f $(docker images -a -q)**

## Cache Images

**docker pull microsoft/dotnet:1.0.0-rc2-core**

**docker pull redis**

**docker pull tutum/haproxy**

# Demo Projects

<https://github.com/SteveLasker/DotNetCoreMultiService>

<https://github.com/SteveLasker/DotNetMultiService.API>

<https://github.com/SteveLasker/WorksOnMyMachine/tree/021-Demo> - possibly merged with master

<https://github.com/vegasbrianc/docker-compose-demo.git> - Scale demo, until we get the RedisCache update for .NET Core RC2

# Docker **build**

**Script**

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***Baseline*** | |
| **Docker run busybox** | Let’s run a container, in this case one called busybox  Notice that it just runs, no errors, but no output |
| **Docker run -it busybox** | The busybox image gives us a basic embedded Linux instance. It supports basic shell commands, but to have it execute those commands we either need to tell it to run them when it starts up, which we’d like want to see, or we can run the container in interactive mode |
| **Ls** | Using linux commands, lets do a few, like get a directory listing |
| **Cd usr**  **Mkdir temp**  **Touch a**  **Touch b**  **Touch c**  **Ls** | We’ll use the touch command to create some empty files  Let’s switch to the usr directory, make a temp directory and create a few files and list them out  All of this is possible, because we’re executing commands in the container, using a ssh connection |
| **[CTRL] + [D]** | We’ll exit the session with CTRL + D |
| **Docker ps** | If we look at the current containers, we’ll see there aren’t any because we exited the container |
| **Docker ps -a** | We can see the stopped containers by listing all |

|  |  |
| --- | --- |
| ***Demo: Registry Search*** | |
| **Docker search microsoft** | We can search the default registry of docker hub for all images that include Microsoft |
| **Docker pull microsoft/dotnet** | Now, this one takes a little longer, and we can see several lines.  Are these just chunks we’re downloading?  Turns out these are multiple layers of a docker image. This is one of those parking lot items that we’ll come back to |

|  |  |
| --- | --- |
|  |  |
| ***Demo: Building Docker Images*** | |
| * Launch PowerShell * CD: **HelloWorld** * DIR to list the empty contents * Launch VSCode:  **Code .** | Now that we showed how to pull and run a basic container, let’s build the most basic Hello World app  We’ll launch PowerShell to execute our commands  We’ll work in an empty directory, just to show a basic Hello World scenario  We’ll use VS Code, also to keep it simple |
| * **Dotnet new** | To prime things, we’ll use the dotnet CLI to create the most basic new app  In VS Code, we can now see the two basic files  Program.cs and project.json  project.json includes our dependencies  Program.cs, well, that includes our program, which we can see is pretty basic |
| * **Create a new file:** CTRL + N * Save as **dockerfile** | We’ll create a dockerfile which we’ll feed to the docker build command in a moment |
| **FROM microsoft/dotnet:1.0.0-rc2-core**  **WORKDIR /app**  **COPY /app /app**  **ENTRYPOINT dotnet HelloWorld.dll** | Let’s look at our dockerfile here  We first have a **FROM** statement which instructs Docker to start with a base image named dotnet, with a tag (version) of 1.0.0-rc2 and we’re using the core version of DotNet  We’ll create a working directory then copy the contents from app on my host (my laptop) to the contents of an app directory in the newly built container  Lastly, we’ll simply tell dotnet to start our dll.  Now, you might also notice we have some docker language services available for syntax coloring and we can even query the registry for the images available |
|  | Now you might be wondering where the app directory on my host came from, and where the DotNetHelloWorld.dll came from. While we could copy our code into the container, restore it and run it, that’s not an optimized image.  What we really want is the compiled output of our dotnet app. This is the results of dotnet publish. |
| **Dotnet restore**  **Dotnet build -c release**  **Dotnet publish – c release -o app** | I’ll type them both to show what happens  First, I’ll build, with the release configuration  Then, we’ll “publish” the release configuration, and place the output in the app directory. Now you’re seeing where the app directory came from on the left side of our dockerfile copy /app /app directory |
| **Docker build –t helloworld .** | We’ll execute the docker build command.  The –t tells docker to tag, or name the image we’re building.  . simply tells docker build to use a default dockerfile in the current directory  We could have entered:  Docker build –t helloworld dockerfile |
| **Docker images**  **Docker ps** | Now that we build an image, we can see it in our list  However, if we look at the running processes, we won’t see it instanced. It’s just built. It’s sitting quietly, waiting to go |
| **Docker run helloworld** | Let’s run this container, based on the image we just created  Let’s just take a second, and digest this. Look at how fast this happened. We type docker run… The container is instanced, and starts executing. We are instancing an entire OS environment, not just an app. Can you imagine starting a VM, having it perform an operation and shutdown? How long would that take?  And, we could run multiples of these |
| Change the body to:  **for (int i = 0; i < 15; i++)**  **{**  **System.Threading.Thread.Sleep(1000);**  **Console.WriteLine("Hello World!");**  **}** | However, since it executes so quickly, we’ll need to do a little extra  Let’s just add a sleep, and have it run for 15 seconds |
| **Dotnet publish – c release -o app**  **Docker build –t helloworld:loop .**  **Docker run helloworld:loop**  Run it several times  **Docker run -d helloworld:loop**  **Docker ps** | Now, I’ll do a little teaser here and start to show the beauty of containers, and how easily we can spin up multiples  We’ll build and publish the app. Notice, publish will automatically do the build for us  We’ll do a docker build, this time we’ll tag it with loop as our image name  Lastly, we’ll run the image, with our loop tag. But, notice I needed to pass the -d parameter, to detach. We can’t run multiples if our powershell session is attached to the docker process  Once we’ve spun up several containers, we’ll do a docker ps to see what’s running |

With that demo, we’ve seen the basic steps for building your app, placing it in a container and running that container. Notice how it just feels like running a process. It just so happens the process includes the entire execution environment. Imagine the possibilities.

# Demo: Volume Mapping

|  |  |
| --- | --- |
| ***Demo: Volume Mapping*** | |
| Open PowerShell | Docker Containers are based on creating immutable images that do processing. Containers can be scaled and instanced. But, the design pattern also assumes they fail. They recover. They may even be moved by the system to maintain density.  The point here is the container, and everything in the container must be disposable.  So, what to do about things you don’t want to dispose? Like log data, data in a database, files that get uploaded, or images processed?  In these scenarios, we use a feature that creates a tunnel out of the container to another location. The tunnel is referred to as Volume Mapping  We’re going to create a volume mapping from the container to a more durable location.  The most obvious would be Azure Storage |
| **docker run -it -v /c/Users/SteveLas:/wormhole busybox** | Let’s create a wormhole from our host (our windows developer machine) to the container |
| **Ls**  **Cd wormhole**  **Touch a.txt**  **Touch b.txt**  **Ls** | We’ll see what’s visible. We’ll use ls to list the directories available  Yup, there’s our wormhole  I’ll navigate into that directory and create a few files |
| **Explorer: Open C:\Users\** | Notice we have a Host directory, that was created when the container spun up and attempted to volume map to its location  And, we now have files, persistently saved |

# Demo: Compose

|  |  |
| --- | --- |
| ***Demo: Docker Compose*** [***MultiService***](https://github.com/SteveLasker/DotNetCoreMultiService) | |
| Open PowerShell  **Code .** | Scenario:  You’ve got a fix to make to a project. It’s just a one-line fix, but is that the hard part?  How many other things do you have to setup, just to make that fix?  You need the project itself. You likely need a database, or a cache, a WebAPI. Now you have to get all those checked out, compiled, running. Do you have all the pieces to make them work?  With Containers, we can spin up that environment quickly. Not just the sourcecode, but the running environment.  Let’s say we need to make a change to a Wordpress codebase  Wordpress depends on MySQL  With Docker-Compose, we can spin up our codebase and our dependencies. We only need to compile the components we actually need to compile. For instance, MySQL is available as an image, so we just need to provide some configuration information  We’ll use VSCode here, again for simplicity. Visual Studio is coming, no worries |
| Open docker-compose.yml | Now, let’s open a docker-compose file  What do we see?  There’s a collection of services – yeah, this is a collection. We just don’t have all the noisy curly brackets and commas to deal with  But, the number of spaces are important, and don’t use tabs. Apparently the yaml specification folks had tab issues, so they’re not allowed |
| **Docker-compose up -d** | Lets see this in action  We’ll run docker-compose up. This is the exe for compose  Since we didn’t pass it a filename with -f, it assumes a file of docker-compose.yml is present in the current working directory.  We’ll pass -d to detach the commandline so we can let the containers run and execute more commands |
| **Browser:** [**http://docker/**](http://docker/)  **Browser:** [**http://docker/customer**](http://docker/customer) | We’ll launch the browser to our local docker host  Voila – we have our .NET Site, and we can drill into customers |

# Demo: Scale

## Setup

git clone https://github.com/vegasbrianc/docker-compose-demo.git .

|  |  |
| --- | --- |
| ***Demo: Scaling an app*** | |
| CD C:\Users\SteveLas\Documents\Demos\Scale | Let’s take a look at a simple app that we’ll scale up |
| Open docker-compose | This is where it gets interesting  Notice we have three services:  Web  Redis  Lb – load balancer  The website runs on port 5000, exposes it, but it’s not actually addressable outside of the docker host. Notice no ports are listed  The redis cache is linked to the web project |
| Highlight the load balancer | We’re using a haproxy base image, notice no dockerfile, as we’re using the baked image, and just configuring it  It’s linked to the “web” service. This is important as there’s some basic discovery that will happen here  The load balancer works over port 80. The container listens on port 80, and it’s exposed on the host as port 80. Which also means we can only have one of these per docker\_host  The last thing that’s interesting is the BACKEND\_PORT and balance parameters  What this tells haproxy is listen to port 80, look for linked containers on port 5000 |
| **Docker-compose up -d** | Lets instance this group.  Notice we only have one instance |
| Browse to: <http://docker> and hold F5 | If we just hold F5, we can see the page refresh and count up  Notice the host name is the same |
| **Docker-compose scale web=3** | If we tell docker to scale the web service to 5, notice the extra containers are running |
| **Docker ps** | The only problem is HA Proxy isn’t actually aware of this |
| **docker-compose up --force-recreate -d** | If we recreate the collection, when the containers restart, we’ll see the HAProxy picks up the additional containers and starts round robin  This is also where we start to see the primitives end, and the orchestration engines are needed.  The reality is our containers may be run on multiple hosts, across a farm. We may scale up, down, automatically. The system should selfheal. If a rack dies, the system should re-instance those containers to another host. As it spins up the additional containers, once running, it informs the load balancers there are new endpoints.  Likewise, when it needs to gracefully remove a container, it first bleeds traffic off the containers to be removed, then once there’s no traffic, it removes the container.  All this is more complicated than an individual API may do. It requires agents on each host, load balancers on each host. Masters/Slaves, etc. |

# Demo: ASP.NET

|  |  |
| --- | --- |
| ***Demo: Docker Compose*** | |
| CD **MultiService\src\MultiService Code .**  Open Docker-compose.yml | We’re going to work with a collection of containers that are already setup in a Compose file  Looking at this docker-compose.yml file, you can see a collection of services:  multiservice – the web front end, which uses an image reference  webapi – the WebAPI service, also using an image reference.  You’ll want to reference images if you’re deploying to production as you never build an image as part of an orchestration solution.  Notice the front end request port 80 to be mapped from the host into to the container.  However, the webapi image has no additional configuration. This says the webapi container will not be publically exposed to the internet, and only accessible within the default network.  We see a depends\_on statement that tells docker-compose/swarm to spin up the webapi container before starting the web front end  Lastly, we see the CustomerAPIService environment variable set to a url that looks kinda strange. If we look close, we’ll notice this base url is the same as the service name for our webapi. This is how docker configures the private dns to resolve the containers |
| **Docker-compose up -d** | By calling up, we’re telling docker to look for a default docker-compose.yml file and spin it up. The -d says start the containers, and detach, letting it run, freeing up the shell. |
| **Start** [**http://docker**](http://docker)  [**http://docker/customer**](http://docker/customer) | We’ll launch the browser to see the app running.  This is a basic ASP.NET Core RC2 app. We can click around, and we can enter a URL for the list of customers that comes from our WebAPI container. |
| **Docker-compose down** | We can test the API directly, by instancing it directly;  We’ll spin down the collection of containers |
| **docker run -d -p 80:80 stevelasker/multiserviceapi**  **Start** [**http://docker/api/customer**](http://docker/api/customer) | Now, we’ll run the wepapi container directly.  Notice how we can call the API directly, and publically because we’ve added the port mapping |
|  |  |