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>

# Demo environment setup & prereqs

## Creating Docker Hosts in Azure

<https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-docker-machine/>

openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout AzureMgmnt.pem -out AzureMgmnt.pem

openssl pkcs12 -export -out AzureMgmnt.pfx -in AzureMgmnt.pem -name "Azure Management Cert"

openssl x509 -inform pem -in AzureMgmnt.pem -outform der -out AzureMgmnt.cer

docker-machine create -d azure --azure-subscription-id="subid" --azure-subscription-cert="C:\Users\SteveLas\.docker\AzureCerts\Feb2016\AzureMgmnt.pem" azurehost-linux

# Reboot Tasks

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

**docker-machine start default**

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

# Demo Reset

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

**docker rmi busybox**

**docker rmi -f wordpress\_web**

**docker rmi -f orchardup/mysql**

**docker rmi -f orchardup/php5**

**docker run -it ubuntu bash**

## Cache Images

**docker pull microsoft/aspnet**

**docker pull Ubuntu**

**docker pull node:0.12**

**docker pull orchardup/mysql**

**docker pull orchardup/php5**

**docker pull python:2.7**

**docker pull tutum/haproxy**

**docker pull redis**

# Network Change

**docker-machine restart default**

# Reset All Images

Only necessary to clear all, and start from scratch.

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

**docker rmi busybox**

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

# Reset Wordpress demo

[**https://docs.docker.com/compose/wordpress/**](https://docs.docker.com/compose/wordpress/)

$ curl https://wordpress.org/latest.tar.gz | tar -xvzf -

**Cd C:\Users\SteveLas\Documents\Demos\WordPress\wordpress**

**docker-compose kill**

**docker rmi -f wordpress\_web**

**docker rmi -f orchardup/mysql**

**docker rmi -f orchardup/php5**

# Docker **build**

**Script**

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| ***Baseline*** | |
| Open a PowerShell script  **Docker run hello-world** | To get started, we first need our Docker Tools – and we can download the Docker Toolbox. This gives us the core APIs we need.  Rather than jump into what they all are, let’s first start with what they can do. I’ll come back and explain them one by one.  First, we’ll call the most basic docker command:  Docker run tells docker to run a container image  In this case, the hello-world image |
| **Docker run busybox** | Lets run another 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  Lets 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/aspnet** | 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 * Note the directory of your demo: NodeHelloWorld * 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 website  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 |
| * **Create a new file:** CTRL + N * Save as **server.js** | We’ll create a basic Node.js app  Let’s create a new file and save it as **server.js**  We’ll save it before typing to get some language service love based on the .js file extension |
| var http = require('http');  var handleRequest = function(request, response) {    response.writeHead(200);    response.end("Hello World!");  }  var www = http.createServer(handleRequest);  [www.listen(8080)](http://www.listen(8080)); | We’ll paste in our basic Hello World Node.js app and tell it to listen to port 80 and save the file |
| * **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 node:0.12**  **COPY server.js /server.js**  **CMD node server.js** | Let’s look at our dockerfile here  We first have a **FROM** statement which instructs Docker to start with a base image named node, with a tag (version) of 0.12  We then copy our server.js file to the root, named server.js  Lastly we tell docker what command to execute to run when it gets started. In this case, we tell the node runtime to run the server.js file  We’ll save the dockerfile  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 |
| **Docker build –t node-hello-world .** | Using PowerShell, or GIT or your CMD window, we’ll execute the docker build command.  The –t tells docker to tag, or name the image we’re building Node-Hello-World  . simply tells docker build to use a default dockerfile in the current directory  We could have entered:  Docker build –t Node-Hello-World 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 |
| **Docker run -d -p 80:8080 node-hello-world** | Let’s run this container, based on the image we just created  There’s a few parameters here:  -d – run the container and detach the command line, letting the container continue to run, while we get our script window back  -p 80:8080 this tells docker to enable the network, routing traffic from the host, on port 80, to the container on port 8080  This introduces an important concept in how important it is to understand the network in a docker environment. We’ll cover this in a moment. For now, lets just remember the outside 🡪 of the port mappings. You’ll often see both sides the same value, and when you need to remap it, it’s confusing if the host is on the left or right.  I remember it as, traffic is starting from the left, just as I’m typing, and running into the container, which is on the right |
| Open the browser to 10.0.75.2 | Lets open the browser and view our app  Notice the URL is NOT our local machine, nor the docker container. It’s the DOCKER\_HOST, which routes traffic to its containers |
| **Docker run -d -p 8000:8080 node-hello-world** | Now, lets do a little teaser here and start to show the beauty of containers, and how easily we can spin up multiples  Notice how I mapped the ports this time. I’m still using port 8080 inside the container (on the right). But, on the left, rather than use port 80, I exposed it on 8000  Because we’re doing NAT between the host and the containers, we can’t expose two containers that are addressable on the same port – but let’s table that for the moment and see our two containers running |
| **Docker ps** | First, we’ll show both containers running with docker ps  Noce the image name and the command are the same. But notice the Container\_D and Ports are different.  Also, did you notice how fast they spun up? Try that with a VM |
| Open another tab to:  192.168.99.100:8000 | Lets go back to the browser and we can see the two instances running  We can refresh both pages and both are responsive |
| **docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/run/docker.sock dockerui/dockerui** | To see all our containers, visuall, we’ll use dockerui, an effort Michael Crosby from Docker has been working on |
| **Docker rm –f** [container\_id] | Just to prove we are running two separate instances, we’ll kill one of them |
| Refresh the browser for the associated port | Notice this one is now unresponsive |
| **Docker run -d -p 8001:8080 node-hello-world**  **Docker run -d -p 8002:8080 node-hello-world**  **Docker run -d -p 8003:8080 node-hello-world**  **Docker run -d -p 8004:8080 node-hello-world**  **Docker run -d -p 8005:8080 node-hello-world** | Lets spin up a bunch of these and show the results |
| **Docker ps** | Of course, we can see these all running in our process list |
| **Docker ps –q**  **Docker rm –f $(docker ps –q)** | Lets clean this up a bit. We want to stop the list of running containers. We could simply type docker rm –f for each container id, or we can use some additional commands  First, we can get a list of IDs using the –q parameter  It turns out docker rm will take a list of container ids, so we can simply send that list to docker |
|  |  |

# 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/Public/Host:/wormhole busybox** | Lets 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\Public** | 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 WordPress*** | |
| Open PowerShell  Cd **WordPress**  **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.  Lets 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 Dockerfile | Let’s look at the Dockerfile.  This is what we’ll need to build our Wordpress container  Pretty basic:  Start with a PHP Image  Add our code  Done |
| 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 |
| Highlight web | We notice the web service and a db service  For the web service we can see a docker build, that uses the default dockerfile, specified with .  There’s the entry point, to start pgp  Port mapping – hmmm, no entering in the docker run – nice!  Volume mapping – this gives us the ability to edit/refresh  Depends\_on – this is where docker-compose will link up dependent containers, in this case the db container |
| Highlight db | If we look at the db service, we notice it’s pretty simple as well.  But the really important thing is notice there’s no build .  It specifies an image, which is already built. I just want to instance the built image, and pass it any configuration information I may depend upon |
| **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://192.168.99.100:8000**](http://192.168.99.100:8000) | We’ll launch the browser to our local docker host  Voila – we have Wordpress running locally. No external dependencies.  How cool is that? |

# 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 app.py | If we look at this python sample, you’ll see it’s pretty simple  We’re using a redis cache to track ‘hits’, and we simply say hello, with the number of hits |
| Open dockerfile | The dockerfile uses a python base image, copies the code, compiles python and starts the python process |
| 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://192.168.99.100> 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=5** | 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 WordPress*** | |
| Open PowerShell  Cd **WordPress**  **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.  Lets 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 |