DotNet from Development to Production Demos

Author: Steve Lasker

<https://blogs.msdn.microsoft.com/stevelasker/>

Table of Contents

[Demo Overview 2](#_Toc474265817)

[Rough Outline 2](#_Toc474265818)

[Key Takeaways 2](#_Toc474265819)

[Getting setup 3](#_Toc474265820)

[Docker For Windows 3](#_Toc474265821)

[Visual Studio Team Services Configuration 3](#_Toc474265822)

[Linux Container Demos 3](#_Toc474265823)

[Cache Docker Images 4](#_Toc474265824)

[Install the az CLI 4](#_Toc474265825)

[Create a Container Registry in Azure 4](#_Toc474265826)

[Create a Container Registry in the Azure Portal 5](#_Toc474265827)

[Create a Container Registry with the Azure CLI 5](#_Toc474265828)

[Configure DC/OS Viewing 5](#_Toc474265829)

[Create an Azure Container Service 5](#_Toc474265830)

[Install Putty 5](#_Toc474265831)

[VS Configuration 6](#_Toc474265832)

[Code Snippets 6](#_Toc474265833)

[Install the Open Command Line extension 7](#_Toc474265834)

[Enable Track Active Item 7](#_Toc474265835)

[Demo 1 (\_\_ min) – .NET Framework – containerizing existing apps 8](#_Toc474265836)

[Section 2 Setup 9](#_Toc474265837)

[Section 2 (\_\_ min) - .NET Core – the single container project 9](#_Toc474265838)

[Section 2.1 (\_\_ min) – Publishing a single container to Azure 15](#_Toc474265839)

[Section 2.2 (\_\_ min) – Setting Environment Variables for Published Apps 18](#_Toc474265840)

[Section 3.1 (\_\_ min) – Testing our CI Build, Locally 20](#_Toc474265841)

[Section 3.2 (\_\_ min) – Configuring CI/CD 24](#_Toc474265842)

# Demo Overview

High-level flow



## Rough Outline

## Key Takeaways

The \_ things viewers should take away from this section/demo will be:



# Getting setup

## Docker For Windows

From here: <https://docs.docker.com/docker-for-windows/> install <https://download.docker.com/win/beta/InstallDocker.msi>

### Visual Studio Team Services Configuration

#### yo team

used to configure VSTS for CI/CD pipelines

Install [Node.js](https://nodejs.org/en/)

npm install -g yo

npm install -g generator-team

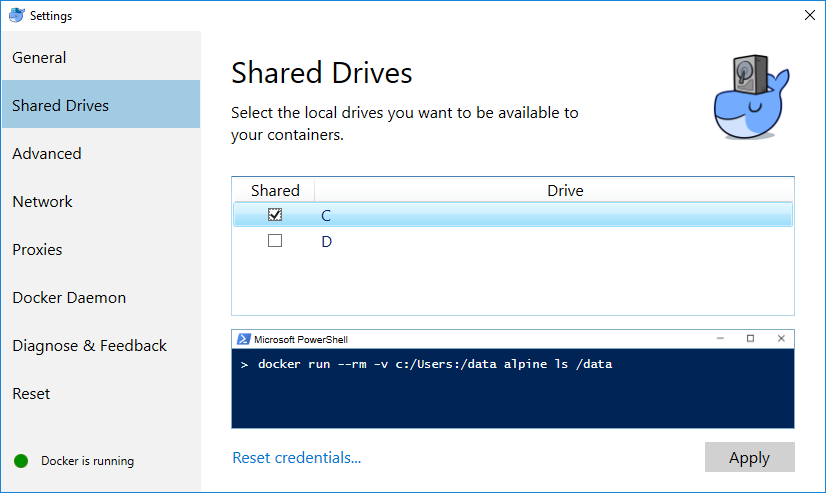
#### VSTS Docker Extension

Add the following extension to your VSTS Account

<https://marketplace.visualstudio.com/items?itemName=ms-vscs-rm.docker>

### Linux Container Demos

Configure Shared Drives – **REQUIRED**



## Cache Docker Images

docker pull microsoft/aspnetcore:1.1

docker pull microsoft/aspnetcore:1.0

docker pull microsoft/aspnetcore-build:1.0-1.1

## Install the az CLI

1. <https://github.com/Azure/azure-cli/blob/master/doc/preview_install_guide.md#windows-cmd>
   1. To install Python, use: [Windows x86-64 web-based installer](https://www.python.org/ftp/python/3.5.2/python-3.5.2-amd64-webinstall.exe)
   2. Check **Add Python 3.5 to PATH**

## Create a Container Registry in Azure

Depending on the audience, you may wish to do this from the Portal or the CLI

### Create a Container Registry in the Azure Portal

<https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-portal>

### Create a Container Registry with the Azure CLI

<https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-azure-cli>

### Configure DC/OS Viewing

<https://docs.microsoft.com/en-us/azure/container-service/container-service-connect>

## 

## ~~Create an Azure Container Service~~

1. ~~Create the Resource Group for the Container Service~~

~~az resource group create -n bikesharing-acs -l southcentralus~~

1. ~~Create the Container Service  
   az --dns-prefix bikesharing -n bikesharing -g bikesharing-acs -l southcentralus~~

## 

## ~~Install Putty~~

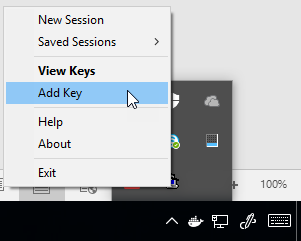
~~Used for accessing the ACS deployed site~~

[~~http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html~~](http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html)

~~use the .ssh folder provided…~~

~~run pageant from the start menu. It will appear in the bottom right systray~~

~~Add a key, using the context menu~~

~~~~

~~Import the id\_rsa.ppk from the~~ **~~%userprofile%\.ssh~~** ~~folder~~

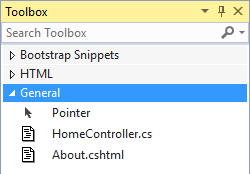
~~This will provide access to browse deployed ACS apps from the marathon UI~~

## VS Configuration

Install Visual Studio 2017

### Code Snippets

1. Clone <https://github.com/SteveLasker/Presentations/> locally
2. Copy <https://github.com/SteveLasker/Presentations/blob/master/DotNetDockerCodeSnippets/NetCore/Controllers/HomeController.cs> to the toolbox. Name it HomeController.cs
3. Copy <https://github.com/SteveLasker/Presentations/blob/master/DotNetDockerCodeSnippets/NetCore/Views/Home/About.cshtml> to the toolbox. Name it About.cshtml

The toolbox should now look like this:  


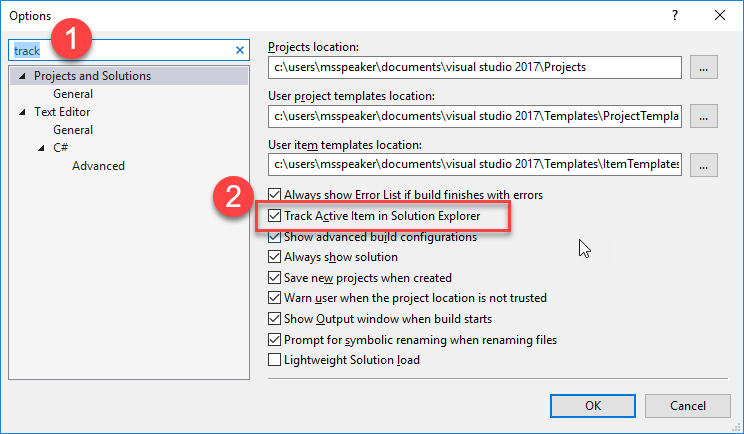
## Install the Open Command Line extension

This is a nice to have as some demos require opening a prompt in the project or solution directory.

<https://visualstudiogallery.msdn.microsoft.com/4e84e2cf-2d6b-472a-b1e2-b84932511379>

## Enable Track Active Item

For the multi-container demo, to highlight which project is being actively debugged, it helps to turn on active item tracking



## Demo 1 (\_\_ min) – .NET Framework – containerizing existing apps

Scenario: You likely have many existing .NET Framework applications that you’d like to containerize for a host of reasons. From improving DevOps workflows to enabling tight density of possibly conflicting resources on the same server(s)

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  |  |  |
|  |  |  |

In just a few mins…

# Section 2 Setup

# Section 2 (\_\_ min) - .NET Core – the single container project

In this demo we’ll start with .NET Core and show how we can start with a single container service we may want to simply deploy with App Services.

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  | * **Create New Project** * **.NET Core** * **ASP.NET Core Web Application (.NET Core)** * Name**: Web** | For the sake of simplicity, we’ll start with a new project, but this would equality work with existing projects. I just want to show there’s nothing up my sleeves so to speak |
|  | * **Web Application** * **[ ] Enable Docker Support** | We’ll start with a Web front end site, but we’ll see how we can add API services in a bit  Notice we have the option to enable Docker right here.  This is great when you know you’re going to target containers. However, I wanted to show what we’d do for existing projects, so we’ll uncheck this here.  We also include reference to the Docker for Windows toolset that enables us to run containers locally and a walkthrough. |
|  | * Replace **HomeController.cs** with toolbox snippet * Replace **About.cshtml** with toolbox snippet | To add some context to where our code is running, we’ll add some content that displays our OS information.  I’ll replace the HomeController.cs with some information regarding our host process |
|  | * **F5** | Just to show there’s nothing up our sleeves, we’ll start debugging. |
| C:\Users\STEVEL~1.RED\AppData\Local\Temp\SNAGHTMLfa16fcc.PNG | * **Click About** | Looking at the about page, we can see we’re running on Windows, on your local machine (HostName) |
|  | * Add **Docker Support** | Now that we see we have a standard .NET Core project running on my local Windows machine, lets add some Docker support. |
|  | * Open **Dockerfile** | This will add several artifacts for us.  The first is a standard dockerfile.  This is scaffolded for you, but you can enhance as you’d like as it’s part of your project. VS won’t replace it.  **FROM** pulls the runtime aspnetcore image that we put the compiled/“published” output into.  **ARG** is a variable VS will set for us that sets the directory by which to copy the published output.  **EXPOSE** tells the container what port the app will be listening on. This is a hint, used by Docker to default port mapping. This port should match your application. You’ll notice we didn’t use 5000 here. That’s intentional. 5000 is used to avoid conflicts with IIS or other processes on your “host”. However, in the container world, each container is a dedicated host, so we don’t need the 5000 workaround, and we didn’t want to carry a workaround into a new environment that wasn’t required.  **COPY** takes the published output, either from the source directory, or a default obj/Docker/publish directory.  **ENTRYPOINT** tells docker what process should be started upon container run. Notice the syntax we use. This allows developers to override the default entrypoint parameters. |
|  | * Open **Docker-compose\docker-compose.yml** | One of the most recent changes we made was to support the concept of a solution based compose project.  As you emerge your development from more basic single container scenarios to multi-container/microservice scenarios, you’ll want to spin up a collection of containers that will all interact with each other, by a named reference.  VS will also use this project and compose file to coordinate the various docker builds needed to build the docker images and start debugging a collection of projects. |
|  | Point**: docker-compose project** set as **startup project** | Before we dig into the details of the dock-compose file, let’s talk about the project.  This docker-compose project gives us more control over the build and debug experience. You’ll notice the docker-compose project is set as the startup project. And the debug target is docker. This means VS will trigger a build of the dependent project, as defined in the docker-compose.yml file and issue a series of docker-compose commands to build and run (up) the collection of containers. |
|  | **F5** | Lets give this a whirl. We’ll start debugging and we can watch the series of commands in the output window.  In addition to the docker build commands, we see some other interesting commands as well. Including docker-compose -f with a more complex docker-compose.yml file. More on this in a bit.  But we also see some docker inspect commands that determine the ports assigned. In the Windows container case, VS will also determine the IP address. |
| C:\Users\STEVEL~1.RED\AppData\Local\Temp\SNAGHTMLff3efea.PNG | **Click About** | With the code we added to the about page, we can see this is running in a Linux OS. Also notice the hostname is no longer your development machine, but a random id for the newly created container.  Also notice the collection of environment variables are much smaller. This is what we start to see when we have a cloud focused/container OS. |
|  | Open **About.cshtml** | Now that we have our debugger running, lets see what we can do.  First, lets do a little “edit and continue” type of editing. |
|  | <h3>@ViewData["Message"]</h3>  <p>here's a live change.</p> | We’ll add a little text here in our about page.  Now, lets just digest this a bit. We’ve got our code here in VS, saved to our local drive. We have a Linux container running, with our code “in that container”. When we save this file, as a VS user, what would use expect? When you consider the complexities of multiple operating systems and containers running, what would you expect? |
|  | Save **About.cshtml** | I’ll save the file and refresh the browser page. |
|  |  | What do we see? Our page is updated. Of course. This is Visual Studio, we’re here to make things easy for you, so you can focus on your development. |
|  | Set a **breakpoint** in **HomeController.cs**  **Refresh** About | But what about debugging? Lets set a breakpoint in the HomeController.  When we refresh our About page, we can see our breakpoint hit, with all the variable information you’d expect from the VS debugger. |
|  |  | But, there are a number of things that had to happen to make this work. We want to provide that VS experience, but we also want to adhere to the Docker experience.  Now that we see this all running, lets lift the hood a bit to demystify the magic. |
|  | Open **docker-compose.yml** | Let’s take a second look at what we have in here:  **services:** are the collection of docker services that will run when you issue a **docker-compose up**. They can be based on an image that you may not have access to the source to, or a project on your machine.  **web**: this is the named instance of our first service.  **Image**: this is the name of the image that will be used. In the case where we’re building the image, based on the build: entry, docker-compose will tag the image with the name we declare here.  **build**: informs docker-compose what parameters should be used to build the image.  **context**: is the directory to copy into the docker host before a build is executed. This is important to understand. If your context directory has a whole lot of stuff you’ll never use to build your docker image, you’re going to pay a penalty of time for docker to copy the context. You can filter this with a .dockerignore file, but more on that later.  **dockerfile**: this simply specifies the name of the dockerfile used to build, which must live within the context directory. This is particularly useful if you’re planning on building several variations. An ISV might build Linux and Windows variations. |
|  |  | What you don’t see in this file is a bunch of debugger info.  Docker has the ability to merge multiple compose files into one. What we’ve done is split up the VS requirements for debugging into a separate file. This allows you to keep your runtime docker-compose file separate from your development content. |
|  | Expand **docker-compose.yml**  Open **docker-compose.vs.debug.yml** | With the understanding that docker-compose will merge these files together, we can review some additional content that’s been added here:  **Build: args:** notice the value for source is added here. Although, it’s really adding the value of an environment variable.  **environment:** You may have assumed the change to the cshtml file just worked because of dotnetwatch. As it turns out, changes made from VS on Windows don’t actually propagate across to the Linux container. By adding DOTNET\_USE\_POLLING\_FILE\_WATCHER=1 we tell the razor engine to poll for changes. And if found, recompile the razor page real time.  **volumes**: these are network drives that are mounted from the Linux container to our development machine. They are pairs. The left side is the host directory, as referenced from the context of the compose file. The right side is a UNC reference, in Windows terminology.  If we look at the entries, you’ll notice your project is sent in, or the published output is sent in. (./web : /app)  We also mount in the nuget package cache. This saves a lot of time during development. Rather than have to download the package cache from the internet each time, we can simply reference the cache VS already has on your development machine. When VS builds, it’s bringing down the packages necessary to build your project. |
|  | Open **%userprofile%/clrdbg/** | The third is the debugger. .NET Core uses a clrdbg debugger, which is saved in your user directory.  By referencing these two directories during development, we can save a lot of time for each container build as they don’t need to be downloaded and unzipped to your image.  While we could build a separate development image that already has this content, that would mean you’re not developing in the production image.  Using these methods, we can preserve a docker image as defined by a single dockerfile, with a production compose file, with the VS content merged in during development time.  This also means the :dev image doesn’t actually have the content to be run directly. |
|  |  | **entrypoint**: As with the image declaration, this overrides the default value. To run the dotnet core debugger, we must start the process under the debugger. We don’t actually attach to an existing process, we must start the process with the debugger. This strange looking value is a Linux pattern to start a process of opening a null file. The container will stay running as the process doesn’t end.  With the container running, VS will execute a docker command to start the debugger.  **labels**: these are additional hints used by VS to determine whether Windows or Linux containers are used for things like forward or back slashes.  By using labels, we can fit VS configurations within the docker-compose supported syntax, and only added in the vs.debug file avoiding entries in the production docker files. |
|  | Open **docker-compose.vs.release.yml** | Looking at the vs.release version, we can see we’re still mounting the debugger, but aren’t volume mounting the code, nor packages. This means this is a production ready image, that we volume mount the debugger to provide debugging, but the image is still a full production image with all the content. |

# Section 2.1 (\_\_ min) – Publishing a single container to Azure

In this demo we’re going to show how we can host a single container in Azure using Azure App Services.

We’re not quite ready to configure a continuous delivery pipeline. We just want to see our app running in the cloud, so our peers or manager can see what we’ve been working on.

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  | * **Right Click – Publish** | We’re going to simply publish this app to App Service Linux. |
|  | * **Click Create Profile** | We can see an updated view of the publishing dialog. Because we have a dockerfile in the project, we’re presented with an additional option to create an App Service Linux publishing profile |
| C:\Users\STEVEL~1.RED\AppData\Local\Temp\SNAGHTML10525aa1.PNG | * **Choose subscription** * **Select TechReady Resource Group** * **Create a new plan** | We’ll configure a few options here.  I’ve already created the resource group I want to put the App Service Plan within. This is just a workaround as the New Resource Group dialog doesn’t ask me for a region, and I do want control over where the resource group is created. |
|  | * **Name the plan** * **Location = Local to you** * **Click Ok** | We’ll create a App Service Plan, which is the management of our App Services within our local region. We’ll put all our azure resources in the same region to keep them network-close.  A standard S1 VM for our App Service plan will usually suffice. |
|  | * **Select an existing Container Registry** | We’ll select a container registry I’ve already created. You can create one right here in this dialog, however it will place the container registry in the same resource group as the App Service. Not normally a great idea as you’ll likely want to maintain a few registries across multiple resources. So, for now, I’d suggest creating registries in the Azure Portal or the az CLI until we can update the create experience. Although, it does make a quick and easy demo |
|  | * **Click Create** | We can now click [Create], which will create the Publish Profile and provision any Azure Resources. For instance, we created a new App Service Plan. Had we created a new Container Registry, it would also be created at this time.  However, it hasn’t yet published our app. We’ve created the Publish Profile against existing, or newly created resources. |
|  | * **Expand Properties** to show the publish profile | We can see the publish profile right here under the properties node. We can create multiple publish profiles for various targets. We might publish to various staging, testing or experiment sites. |
|  | * **Click Publish** | Now that we have the Azure resources and the profile created, we can publish to our Azure App Service.  Visual Studio will go through the various steps to:   * Compile the project, in Release mode * Build the production Docker Image, locally using the Dockerfile in the project * Push the docker image to the registry, which you’ll notice the docker push command pop up. * Configure App Service Linux to pull and run the image |
|  | * **View in Browser** | After a few moments, we’ll see our browser pop up with our content running in Azure App Service. |
|  | * **Open About.cshtml**   <h3>@ViewData["Message"]</h3>   * <p>Running in Azure App Service</p> | With the publish profile primed, we can make additional changes  Let’s change the about page a bit. |
|  | * **Right Click-Publish** * **Click the [Publish] button again** | We’ll republish the app again to see our change.  This time, we don’t need to go through the additional steps to create the profile, we can just publish with the existing profile  Also notice the Docker Push runs much faster. Since the registry already contains the base layers for ASP.NET and the underlying debain image, we just need to push the content layer that makes up our app. It’s not the delta of the change. But rather the content in the COPY command within our dockerfile. For a new .NET Core project, that’s only about 10mb. |
|  | * **Click the About Page** | Once the image is pushed and re-pulled in Azure App Service, we can see our change.  Pretty easy, huh?  But, what happened to our OS, FX and Environment Variable debugging information? |
| if (Environment.GetEnvironmentVariable("ASPNETCORE\_ENVIRONMENT") == "Development") | * **Open HomeController.cs** | Notice all our debugging information is wrapped in an if statement.  Since our production environment doesn’t declare this environment variable, it’s not included.  But we can change this in a few ways.  Lets explore those. |

# Section 2.2 (\_\_ min) – Setting Environment Variables for Published Apps

When working with containers, we define all the generic content/code within the container. We include deployment specific variables, but don’t include deployment environment specific values. Or, we may have some defaults, but we never want to re-build an image because it must be deployed a second time in the same environment, or to another environment.

For example, the same web image should be run in dev, staging and production. Within any environment, we can instance the same image hundreds of times. Some values such as an ip address, port, hostname are automatically dynamically assigned by the docker host. Our app code should work the same way.

But some information like database, azure storage account, telemetry capturing the container should attach to must be externally provided. Let’s start with something simple like the ASPNETCORE\_ENVIRONMENT value. We’ll show how we can set the default value in the container and show how we can override it in the hosting environment.

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
| FROM microsoft/aspnetcore:1.0  ARG source  WORKDIR /app  EXPOSE 80  ENV ASPNETCORE\_ENVIRONMENT=Development  COPY ${source:-obj/Docker/publish} .  ENTRYPOINT ["dotnet", "Web.dll"] | * **Open Dockerfile** * **Add ENV ASPNETCORE\_ENVIRONMENT=Production** | We’ll start by adding a default value, which is baked into our docker image. |
|  | * Set to Debug * F5 | If we start debugging and view the About page, you’ll notice we still see all the information within the if statement. |
|  | * Open **docker-compose.override.yml** | You may be wondering why we see the “development” info while running locally.  This isn’t “magical”. Magic just means you don’t know the real answer.  When VS issues a docker-compose up, it merges in a few docker-compose files. [By convention, Docker will merge in a docker-compose.override.yml file](https://docs.docker.com/compose/extends/) without specifying -f  Looking in this file, we see a few additional values:  environment:  - ASPNETCORE\_ENVIRONMENT=Development  This sets the environment variable for this particular instance. Meaning it’s not baked into the image, rather set when docker-compose up is executed, which does a docker run setting the variable  ports:  - "80"  This one is more interesting. By providing a single value, it says the container has some traffic on port 80. Please assign a dynamic port on the host and route (NAT) it to this port on my container.  Since this value is set at instance time, it would override anything in our image. Meaning, it would override the value in the dockerfile we just set.  However, when our image is run in App Service, it doesn’t use these compose files. |
|  | * Right Click**-Publish** * Click the **[Publish]** button again | To untangle this a bit, lets simply publish our app to App Service and see what happens.  We’ve set the environment variable to production. Our if statement shouldn’t show anything. Even tho we see this information during debugging within VS as VS uses docker-compose which will merge in our override file. |
|  | * Click **About** | Once published, we see our About page is pretty clean. |
|  | * Open the Azure Portal * Select App Services * Select [your] App Service and click it to enter configuration settings | Now that we’ve seen how we can bake a default value within the image, let see how we can override that value in with **App Services 🡪 Application Settings** |
|  | * Click Application Settings * **Key** ASPNETCORE\_ENVIRONMENT * **Vale** Development * Click Save | In the configuration of our App Service, we can see some application settings. There’s an additional Docker Container configuration node, but we’ll ignore this for the moment as we’re looking to set some environment variables which are stored in Application Settings.  By adding our name/value pair in the App Settings section, we’re telling App Services to set these environment variables.  When configuration information is changed, App Services will pull the image again and re-run it, so this will take a moment. |
|  | * **CTRL F5** in the browser to refresh the browser | With this change, we can now see the debug info, including the ASPNETCORE\_ENVIRONMENT variable that’s set. |

In this demo we’ve seen how we can create configuration information that’s extracted from our code. Using environment variables we can set default behavior, override it while debugging and set environment information on the host the image is instanced. We’re using standard docker conventions to provide the experience you’ve come to expect from Visual Studio.

Lastly, we demonstrated this by publishing our app from our local environment to Azure using Azure App Services.

# Section 3.1 (\_\_ min) – Testing our CI Build, Locally

Our next step is to configure CI/CD of our application to Azure Container Services. However, before we blindly send our source off and hope the configuration will work, only to realize we’re missing some silly configuration.

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  | * Open **docker-compose.ci.build.yml** | Before we configure the server to build our project, we’re going to verify it ***can*** be built locally.  Now, you might say, \_\_\_ VS did just build the project… But, what we’re going to do is build the project in VSTS within a Linux container. We want to build the project, restore packages and publish the compiled output without installing VS. This is one of the big advantages of containers, as well as .NET Core.  We’ll now poke into the build definition added to our compose project. |
|  |  | The first thing we must realize is .NET is a compiled language. Just like Java, Go, C++  To create an optimized Docker Image, we only want to put the minimal content to minimize the size of the image as it’s copied across the network.  We also want to make sure our production image starts up quick. We’re not interested in dynamic compilation as we’re not going update the content of the image while running. For any update, even a javascript file or cshtml file, we’ll create a new docker image, test it, and redeploy the updated and tested image. |
| services:  ci-build:  image: microsoft/aspnetcore-build:1.0-1.1 | * Open <https://hub.docker.com/r/microsoft/aspnetcore-build/> * Click on Dockerfile | If we look at the image defined, we see a build image.  Looking at this on Docker Hub, we can see the dockerfile used to build this image  Notice how we configure the environment and the image.  We set an environment variable to tell dotnet restore to ignore the xmldoc files. These turn out to be quite big. I don’t think anyone is doing intellisense discovery in our build image.  You can see certs, node and various tools being installed |
| volumes:  - .:/src |  | The next entry we’ll see is mounting the current directory . with a /src directory in the container.  This make our source available to the container. We could copy it in, but that would take more time. We’re not building a docker image, we’re simply running a docker container to build our code. |
| working\_dir: /src |  | Here, we’re just setting the working directory. The next set of commands will run in the context of this directory |
| command: /bin/bash -c "dotnet restore ./TR-Web.sln && dotnet publish ./TR-Web.sln -c Release -o ./obj/Docker/publish" |  | This is where the real work happens. With the environment setup, we can now run restore and publish on all the projects in the solution.  If you look close, you’ll notice the publish command has an output parameter of ./obj/Docker/publish  This is using our current working directory (/src) and puts the output back on our development machine.  Lets see this at work: |
|  | * Open the **Web project in Explorer** * Open the **obj** folder | First, lets look at the disk and notice what we do and don’t have.  We have our project and source  There’s an obj directory, but no docker sub folder.  Lets build our solution, in a container, right here on our machine |
|  | * Open **Solution in PowerShell** * docker-compose -f .\docker-compose.ci.build.yml up | Using the handy dandy mad mads Command Line extension, we can open powershell directly from VS.  We’re going to use docker-compose to run our build container, with the configuration we’ve got in the …ci.build.yml file.  Notice we’re just using docker-compose, but we have to optionally pass in our custom compose file. |
|  | * view the **obj** folder | Now that the build image has done its job, with the volume mounted folder, we can now see the published output. This is all ready to be placed into our docker image.  This is an important realization. We haven’t yet built the image. We just compiled the content, with a docker container to prepare for our docker build. |
|  | * cd tr-web | Now, we’ll build our image. We first need to navigate into the web project folder. |
|  | * Open **dockerfile** | From here we can now build our docker image.  Since our dockerfile has a copy command that defaults to obj/Docker/publish the docker build will take the compiled output we just created. |
|  | * docker build -t web:test . | We’ll tag the built image web:test and we’ll specify . to use the current directory as the build context. Docker will look for a file named dockerfile by convention. |
|  | * docker images | We can now see the image we just built.  Also notice the image sizes.  Our web:test image is ~292mb  The base aspnetcore:1.0 image is ~274mb  and  The aspnetcore-build image 1.33gb  We can see that we might use a big image to compile our app, and we optimize the runtime image. |
|  | docker run -it -p 8080:80 web:test | Finally, we can run our image to see how it works.  I’ve told docker to run it interactively (it)  Take the container port 80 and host it on my docker host as port 8080. I remember it as reading from left to right. The host routes traffic to the container. 8080 🡪 80  Notice the .NET Core console output says it’s in production mode as we don’t have the development value set  Also notice that the container is listening on port 80. However, the docker host has mapped it to port 8080 |
|  | * browse <http://localhost:8080> * Click around | If we open the browser, we can now navigate to our running container.  We can see our about page shows limited information, as we’re in production mode. |
|  | * CTRL + C * Refresh the browser | If we kill the container, we can see we’re no longer able to view the website. |

# Section 3.2 (\_\_ min) – Configuring CI/CD

Now that we know we can compile, publish and build a docker image, lets configure VSTS to do this remotely.

|  |  |  |
| --- | --- | --- |
| **Visual** | * **Demo Steps** | **Talking Points** |
|  | * Open **Solution in File Explorer** * Click **Add to Source Control** | To configure Continuous Delivery, we of course need source control integration.  If your project doesn’t have a .git folder in the root of the solution, VS will prompt you to Add to Source Control.  Notice we don’t yet have a .git folder.  Once we Add to Source Control, VS will create .git entry  This enables offline/local source control management |
|  | * Click **Publish Git Repo** under Team Services | We now have the option to place the project under VSTS Source Control. Since our CI/CD will be based on VSTS, this is a good place to start. |
|  | * Choose your **subscription** * Choose the **Team Services Domain** * Name your **project TRWeb Note:** Web is a reserved name * Publish the **repository** | Here we can confirm which Visual Studio Team Services subscription we’ll use for our VSTS configuration  We can then choose the team services domain. This is the root URL your VSTS projects will live in. Remember, VSTS can store multiple projects under a single VSTS domain. In our case, we’ll place the Web, API and other projects under this single domain.  Lastly, we’ll name our specific git repo  With all these set, we can publish the repo from our local/offline git store to VSTS. |
|  | * on the web project: Configure Continuous Delivery | You might think you have to goto VSTS and run through a bunch of steps, answer a whole lot of questions, and get lost for hours reading blogs and docs.  Not the case. We have an experimental extension we’re working on to make this far easier. We have lots of context about your project and your server, right here in Visual Studio. |
|  | Verify/Populate   * Subscription * Registry * ACS instance | To configure a build/release pipeline, we need to know a few basic pieces of information  Which User Account are you using?  We know the VSTS Team Project and Git Repo  What Azure subscription are you going to use, which you might have several for that single login  When the build system builds docker images, where should they be pushed. If you don’t have a registry, you can provision one right here.  Now, I did provision the container registry and ACS/DC/OS cluster already. Registry creation is as fast as creating a storage account. However, DC/OS can take a bit to get up. |
|  | Press **[Ok]** | I should also point out how we think about the various orchestrators as we now have a 3.  Swarm and Kubernetes are the places we see most gravitating to as they’re both designed for the common API/Web App scenarios. They have service discovery, secret management, service definitions and will both support Windows and Linux containers.  DC/OS is a powerful orchestrator. It’s great for internal data processing scenarios, as you’ll see, getting an external endpoint is a bit difficult for it as it wasn’t optimized for those scenarios. But, scheduling chron jobs and such are great.  What you should see is us adding Kubernetes and Swarm to our common productivity tooling scenarios. |
|  | Open VSTS | Once our configuration is complete, we can navigate to VSTS to see our build and release configurations. |
|  |  |  |
|  |  |  |