Lift, Shift, Modernize ASP.NET FX apps w/ASP.NET Core APIs

|  |  |
| --- | --- |
| Role | Individuals |
| Owners | Steve Lasker |

Table of Contents

[Overview 1](#_Toc488147428)

[Demo Bits and licenses 1](#_Toc488147429)

[Client Machine 1](#_Toc488147430)

[Docker For Windows 1](#_Toc488147431)

[Demo Reset 1](#_Toc488147432)

[Demo 1 – Create an MVC ASP.NET FX App 1](#_Toc488147433)

[Demo 2 – Modernize with .NET Core 1](#_Toc488147434)

# Overview

# Demo Bits and licenses

## Client Machine

Windows 10 Client

## Docker For Windows

<https://download.docker.com/win/stable/InstallDocker.msi> Stable build

# Demo Reset

* Docker rm all old images
* Open the \_demoAssets folder

# Demo 1 – Create an MVC ASP.NET FX App

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  | Create New ASP.NET Web App (.NET Framework)  PUT IN THE ROOT OF C  Bugs in Windows Containers prevent the ability to mount volumes from the C:\Users directory  Project: **Web**  Location: C:\Demos  Solution Name: LiftShift  Be sure to use Web as the project name for copied in files | We’ll start with an ASP.NET Web App  I’ll use MVC, but you could do this with Web Forms. The point is it’s an existing app that we’ll want to modernize with additional services |
|  | Choose MVC |  |
|  | F5 | We’ll start debugging, and you’ll see the normal nuget restore, build and debugger loading |
|  | Add Docker Support | To containerize this, we may wonder what would we need to do.  I know by reading about docker I need a dockerfile. But what’s the contents of the DockerFile? Will I still get VS debugging?  No problem, VS helps you, with standard docker assets |
|  | F5 |  |
|  | Open Dockerfile | Look, VS has added a dockerfile.  The image is an ASP.NET 4.6.2 based image. Visual Studio picked this up from the csproj metadata.  The ASP.NET Image is based on the IIS Image which is based on the Windows Server Core Image.  VS uses the optimized image approach. Meaning, we don’t take the source and just put it in the container. We take the published output. So, only the necessary content is put in the image |
|  | Highlight Copy | If we look closely, we see a coalesce copy. If the build arg is empty, we’ll simply copy the published output.  At F5 time, we’ll set the source to an empty directory and volume mount the content in, but lets see it working first |
|  | Open Docker-Compose.yml | We also scaffold in a compose file.  Docker-compose.yml is used to specify the instancing parameters so we don’t have to specify everything to docker run. This is also where we can instance additional containers. |
|  | F5 |  |
|  | Navigate in the browser | We can now see our container running |
|  | Powershell – Docker ps | If we switch over to a command prompt and run docker ps, we can see our running containers |
|  | Open Views\Home\About.cshtml  Add some text  <p>here we are</p>  Save | If we add some content to one of the razor pages, we can see it in our container. We don’t have to rebuild the image and re-instance it. |
|  | Docker inspect [id]  Search for Mount | How does this work?  If we run a docker inspect, we can see we’ve volume mounted the content into the container. This is where we do some standard industry tricks to leverage the same dockerfile, so we don’t have different base images for our development and runtime.  We simply place the content into the container, through a volume mount.  Now, as we make changes, those changes are in the container.  We can also see we volume mounted the debugger in, so we don’t have to copy it into the container, making the container larger or with more of a surface area than we want. |
|  | Open Solution in Explorer  Open obj\Docker  Open docker-compose.vs.debug.g.yml in VS Code | To see how this got added, we’ll navigate to the root of the solution in explorer  In the root we see an obj\Docker folder  Notice we have two docker-compose generated files. These container the VS specific configurations we need for debugging.  We can see our Arg being set to an empty directory  And the root of our project into the wwwroot directory  And, the remote debugger |
|  | Stop Debugging | That’s great to get my code in a container. But we have new work we want to add. We would like to take advantage of .NET Core, or modern, optimized OS images |
|  | Docker images  Docker history [image id] | If we look at the history of the image, we can see the base image is quite big. That’s because the Windows Server Core image is intended to be compatible with the last few decades of technologies that have been developed.  What we want for a go-forward stack is an optimized runtime and an optimized OS – That’s Windows Nano server |

# Demo 2 – Modernize with .NET Core

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  | Add a New ASP.NET Core Web App  Project: **Api**  Location: **DEFAULT** | We’ll add a .NET Core Web App |
|  | Choose Web API | We’ll chose the Web API template as we don’t need all the pixel painting stuff, just the services  You might notice we can add Docker Support here, but we’ll add it on the project to show the “brownfield” scenarios as well |
|  | Open docker-compose.yml | Now, at this point we have two projects. A website and an API.  How would you expect to deploy these?  In the VM world, you might put both sites on the same VM, or put them on different VMs.  In the docker world, we create separate images for each process. So, each is its own docker image.  But, we want to instance them together.  If we look at the docker-compose project and file, we can see one image/service is declared  Lets see what happens when we add docker support |
|  | Add Docker Support  Choose windows | We first see a choice for Windows & Linux as .NET Core supports both. In this case, we’ll stick with Windows. |
|  | Confirm updating… | Since we have the docker-compse.yml file open, VS is just confirming it can be updated with the new service. |
|  | Dockerfile | Here we can see a new dockerfile has been added for our .NET Core Web API service.  If we look a bit closer, it’s not using IIS. .NET Core can host services more natively with Kestrel. All we need to do is specify the dll we’ll start. The dll is the compiled/published output.  Multistage Dockerfiles – Now, this will get a bit more complete when we add multi-stage dockerfiles coming n the near future |
|  | Open docker-compose.yml | If we switch to the compose file, we can now see our second service was added. This is how VS knows how to instance multiple services on F5 as our compose project is our startup project |
|  | Drag in \_demoAssets\Api | Our next step is to make some changes to the API to do something interesting.  We’ll cheat a bit here and just drag in some content to our API |
|  | Open Api\Controllers\Magic8Ball.cs | I’ve added some random answers to our controller.  We just need to call them from our Website |
|  | Drag in \_demoAssets\Web | We’ll add the MVC content to simply call our API |
|  | Open Web\Controllers\Magic8BallController.cs | Here’s where we can start to see the benefits of Docker and the container orchestrators. Rather than have to figure out bigger network discovery issues, we know that containers and services are constantly being spun up. Docker has built in network discovery into its primitives.  So, all we need to do is reference the service by the service name in the docker-compose file |
|  | Open Web\Dockerfile  Paste in:  RUN reg add HKLM\SYSTEM\CurrentControlSet\Services\DNS\Parameters\MaxCacheTtl /v 0 | We’re going to make one small change to our dockerfile as we can hit DNS caching issues.  It turns out Windows is optimized for things being stable. However, we’re going to keep re-instancing our Api service. Each time, it can get a new IP address. This is actually a bit of a bug, but we’ll do this trick to fix it |
|  | F5 | We’ll launch our collection of containers to see our services communicating together |
|  | Click About | The about page is local to the Web Container |
|  | Open Web\Controllers\Magic8BallController.cs  Line 44: HttpRequestMessage request = | If we navigate to the Magic 8 Ball page, we’ll see our Web call our API |
|  | Open Api\Controllers\Magic8BallController.cs  Line 20:  int rInt = \_r.Next(0, answers.Count() - 1); | We’ll also set a breakpoint in our API when called |
|  | Click Magic Ball | As we click the Magic Ball link, we’ll see our calls go through the web front end, and into the API |
|  |  | Lets think about this. We’re just debugged a multi-container cluster of services running two different stacks.  An ASP.NET Framework app using Windows Server Core  And an ASP.NET Core Web API service on Windows Nano  Over time, we’ll be able to debug across Windows and Linux as well. |
|  |  |  |