

VB6 to Docker

By [John Browne](#)

Extinction is normal

The word "extinction" makes me visualize a couple of dinosaurs, seeing a huge fireball in the sky rushing toward Earth. One says to the other, "Wonder what the hell that is?" The other one says, "Ah, probably nothing."

I know a guy, call him Will, who worked at Kodak when they made the first digital cameras. My friend Will, kept running around Rochester saying that this was a hair on fire moment, because digital cameras directly threatened the film franchise that Kodak was built on. Kodak management--whose predecessors literally invented the consumer camera industry--ignored all the warning signs. Digital, they said, was inferior to film. Digital, they said, was expensive. There were no digital cameras, but there were billions of film cameras (now available at your local flea market).

Digital will never replace film.

Right.

Cars will never replace buggies. Email will never replace snail mail. PCs will never replace mainframes. Ecommerce will never replace retail. The Cubs will never win the World Series.

Running in fear

When the iPad appeared, PC manufacturers were terrified tablets would replace PCs (they didn't). When smart phones appeared, camera manufacturers were terrified they would replace actual cameras (they haven't).

When avocado toast appeared, everybody in their right minds said, "yuck."

Change--change causing extinction--used to take awhile:

- The first (steam-powered) automobile was built in 1769.
- The first production automobile with an internal combustion engine was built 116 years later (1885). A handful were built.

- 42 years after that, Ford had produced 1,000,000 Model Ts using the concept of the assembly line.
- By 1920 the number of automobiles in the US had eclipsed the number of horse-drawn buggies.
- Even so, in World War II (1939-1945), Germany and the Soviet Union employed 6 million horses for transport, mechanized transport being the exception rather than the rule.

Change got quicker:

- 1946: ENIAC (first general-purpose programmable electronic computer)
- 1968: First microprocessor
- 1975: Bill Gates and Paul Allen go to MITS in Albuquerque to demonstrate BASIC running on a microcomputer
- 1981: IBM PC announced
- 1989: Tim Berners-Lee invents world wide web so we have a place for cat pictures
- 1995: I'm sitting in the middle of nowhere (Eastern Washington state, desert and cattle ranches) in a diner. Booth behind me are two old guys wearing Carhartts, ball caps with fertilizer company logos, eating lunch. And I can hear them. And they are talking about *Windows 95*. That's when the lights went on for me. Computing, I realized, is now ubiquitous.

And quicker:

- 1994: First smartphone introduced (Simon Personal Communicator)
- 2007: iPhone introduced
- 2018: 36% of world population has one to watch cat videos.

Hey, John, are you rambling or should I care?

This is a blog series about Docker, although that might seem a reach so far. Let's bring it home.

If you're reading this and you're not an immediate family member, then I have to assume you have legacy code. And that code is for a desktop application built with VB6, PowerBuilder, or even .NET/Winforms. And that application is old, big, complex, with at least one database behind a lot of forms and business logic that--over time--has been sorted out. The app works, people use it, and it's a huge liability.

It's a liability because you've built a wonderful factory powered by steam and everyone else is using electricity. The people who work on steam power are getting older and older and are hard to find and don't work cheap anymore. And there's the daily risk of the damn boiler exploding, taking the whole mess with it. You could build a new electrified factory next door but it would bleed you dry and take forever. You could try to replace all the stuff in the existing factory but there are pipes running everywhere and valves and gauges and switches and signs that say "Never touch this!" and you don't know if trying will wreck everything.

Basically you are faced with an existential threat.



Ahoy!

Here's how to keep alive. This series of blog posts is going to take a steam-powered application (ok, VB6) and convert it to a modern web application with RESTful

endpoints, an Angular 6 client UI, ASP.NET Core server components, and readable code. Then we're going to put our whole web app into a container--Docker for Windows--and build it and run it locally in the container. Finally we're going to push the container to Azure and run it from there.

This is like replacing the steam-powered factory with one using cold fusion, with all new machines. And it's going to be less risky, costly, and lengthy than any alternative.

Why you should care about Docker

Docker is the candle on the icing on the birthday cake. The cake? That's a modern web app, written in languages like C#, HTML, and TypeScript. The icing? Continuous integration, continuous deployment (CI/CD). These two alone are the (potentially) extinction-level changes sweeping contemporary software engineering. The meteor that wiped out Waterfall crashed a long time ago. Agile/scrum replaced it, and with that model came the need for better faster methods to build, test, and deploy. Web apps no longer depended on updates delivered via physical media or downloaded msi files; they could be slipstreamed into production systems at will. This created a new set of problems, challenges, and opportunities. It's the new growth rooted in the composting bodies of dead dinosaurs.

Docker just makes it a little easier. But we'll get to that later.

All aboard!

So let's do this thing. Many Bothans didn't die to bring you this information, but I spent more time than I'd like to admit--plus a little bit of my soul--figuring all this out and I'm determined to share it with someone. Fortunately I have colleagues smarter than me who were enormous help (talking to you, Mauricio).

So grab your favorite app, or use our sample code, and [follow along as we take our magical mystery tour of desktop to Docker](#).

VB6 to Docker Part 2

Note: this is part 2 of a four part series. If you stumbled in here from the street, [here's part 1](#) to help get you oriented.

If you want to follow along, [you can download the code here](#).

What's Docker?

[Docker](#), in case you've been hanging out in a deep-water submersible with no internet connection for the past year, is a set of tools to create, manage, and run containers. What, you may ask, are containers? I truly have no idea, but I refer you to [this Wikipedia article](#), which is fairly confusing. I like to think of them as lightweight virtual machines, except that they're not, actually, virtual machines. But from a standpoint of application hosting, they are similar to VMs...you can set them up as autonomous units with their own OS and app infrastructure--they can run inside any host (so you could have a Linux container running in Windows, or vice versa)--and they can access network ports, so you can get in and out. Unlike VMs, containers can spool up very quickly to make it easier to scale up or scale down a workload. And--perhaps best of all--you can build and run them on your desktop, then just move the whole lashup to a different host (like AWS, Azure, or a private datacenter) without worrying about something in the config script that breaks it. If it runs locally, it will run remotely.

It's helpful to keep in mind that a container differs from a VM in that it (the container) only loads the bare minimum to do the job you've asked it for. When you set up a VM, you have, somewhat obviously, an entire (virtual) machine at your disposal. You have a complete OS with all its available services, all the I/O of the machine, full network stack--everything.

But in a container, you only load what you need. So you don't get a full OS, only a kernel. You only get the OS services that have been provided in that kernel--for example, no UI framework. It's intended to be super lightweight, quick loading, and robust from external attacks (you can't exploit what isn't there). For that reason, you have to be thoughtful about what you use as a base image (typically the kernel) to ensure you get all the services you actually need. Of course, some additional necessary services might be installable as part of the container build step.

Docker for Windows

Docker was initially created exclusively for Linux, and relied on a number of Linux bits and pieces. However, since 2014 Microsoft has been steadily embracing both containers as a concept and Docker as a solution. We're going to use Docker for Windows in this demo/tutorial to containerize an ASP.NET Core web application.

Basic workflow

For this demo, we're going to start with an ASP.NET Core web application--our cannonball WebMAP demo app--and run it in Docker. The app uses ASP.NET Core running inside .NET Framework, with Angular 6-based client/front end. [You can learn all about the unique, simplified architecture we have pioneered here.](#)

Here are the steps we'll walk through:

1. Install Docker for Windows
2. Enable Docker support on Visual Studio
3. Create a dockerfile in our application
4. Publish the app
5. Build the container instance with the published app
6. Run the container and verify the app
7. Push the container instance to Azure to run it from the cloud

Install Docker for Windows

Before we can use Docker we have to install it. Since we are running on Windows, we need to [get the Docker Community Edition \(CE\)](#) from the Docker Store. To install it you need Windows 10 64-bit Pro, Enterprise, or Education. Sorry, it won't run on Windows 10 Home. And here's where I hit my first hiccup: I have VMware Workstation on my laptop. Docker uses Hyper-V (Microsoft's hypervisor) and VMWare doesn't. And you can't have two hypervisors in Windows at the same time. So to run Docker you will have to go into the Windows settings and turn on Hyper-V, but then you can't run VMWare VMs. Note: there are some home-grown solutions like a boot option. Or running Docker for Windows in a VMWare VM, which kind of makes my head want to explode. Google is your friend for this one.

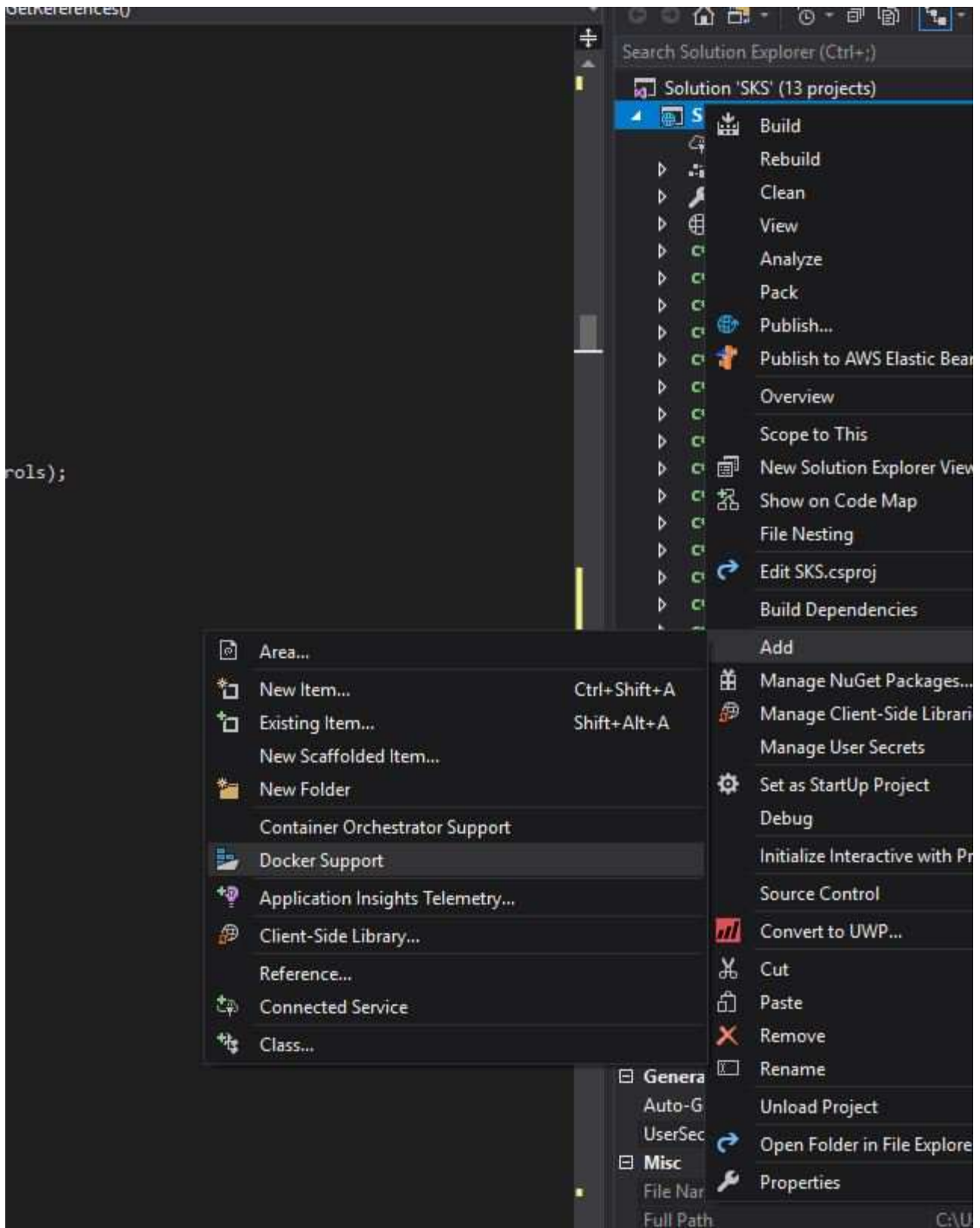
You can set Docker to start automatically or just when you need it. Note it is a Windows Service, so you will find it in the system tray.

To begin, you really only need to do two things:

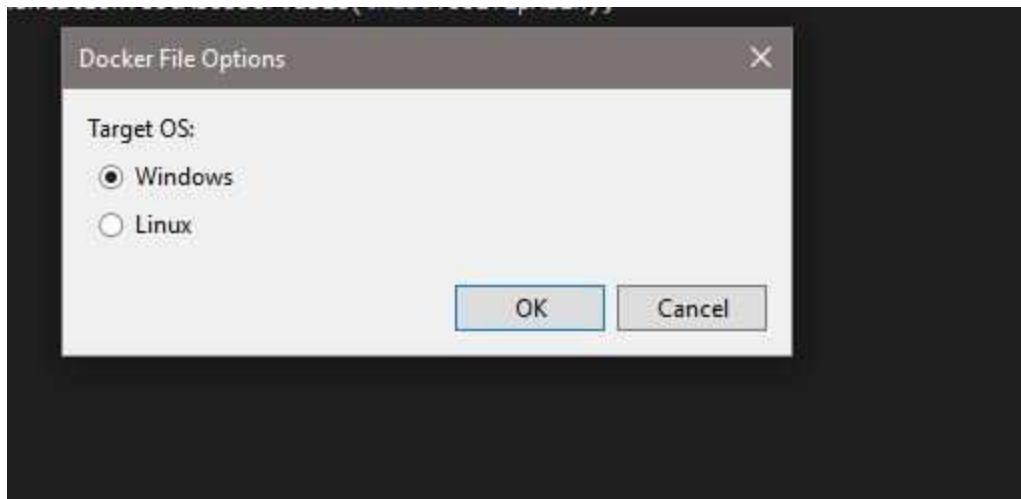
- Right click the Docker icon and select Windows containers (the default is Linux). To run a Linux container, you'll need to switch back. You can only run one kind of container at a time.
- Fire up PowerShell and start using Docker CLI commands. We'll get into that shortly.

Enable Visual Studio support for Docker

You can add Docker support for existing apps via the Add... context menu item on any VS project.



We have to choose between Windows and Linux.



When we do this for our example ASP.NET Core web app, we get exactly two changes: a new file ("Dockerfile") and a .dockerignore file (which we don't need to worry about). Let's look at the Dockerfile:

```
Dockerfile  X frmOrderRequest.X6+fi9cK.generated.cs
1  #Depending on the operating system of the host machines(s) that will build
2  #For more information, please see https://aka.ms/containercompat
3
4  FROM AS base
5  | WORKDIR /app
6  | EXPOSE 80
7
8  FROM AS build
9  | WORKDIR /src
10 | COPY ["Upgraded/SKS.csproj", "Upgraded/"]
11 | COPY ["Stubs/Stubs.csproj", "Stubs/"]
12 | RUN dotnet restore "Upgraded/SKS.csproj"
13 | COPY . .
14 | WORKDIR "/src/Upgraded"
15 | RUN dotnet build "SKS.csproj" -c Release -o /app
16
17 FROM build AS publish
18 | RUN dotnet publish "SKS.csproj" -c Release -o /app
19
20 FROM base AS final
21 | WORKDIR /app
22 | COPY --from=publish /app .
23 | ENTRYPOINT ["dotnet", "SKS.dll"]
```

Now this dockerfile is basically useless. For one thing, it doesn't specify a base image--the necessary OS elements that are needed to run our container. Note that Microsoft understands this--there are too many options for Visual Studio to know what base image you will need, so the comment points us to [this web address](#), which unfortunately isn't any help either. Microsoft has put a large number of [images on the docker hub](#), but it can be tricky to figure out which one to use. Basically you make a guess, build your container, and see if it works. If not, try a different base image.

The real Dockerfile

To get our SKS demo app to run in a container, we will need ASP.NET and Windows Server Core. After some work by one of my more brilliant colleagues, we found that this base image works well:

```
FROM microsoft/aspnet:windowsservercore-10.0.14393.693
```

When we use the Docker Build command, Docker will begin by pulling this image from the Docker hub--this can take some time initially because it's a large image. Here's our dockerfile that will build a container where SKS can run:

```

1  # escape=`
2
3  FROM microsoft/aspnet:windowsservercore-10.0.14393.693
4
5  ## Install the dotnet hosting pack
6  SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop';"]
7  RUN `
8      New-Item -Type Directory C:\Setup | Out-Null ; `
9      Invoke-WebRequest 'https://download.microsoft.com/download/6/E/B/6EBD972D-2E2F-41EB-9668-F73F5FDDC09C/dotnet-hosting-2
10     $process = start-process -Filepath C:/setup/dotnet-hosting-2.1.3-win.exe -ArgumentList @('/install', '/q', '/norestar
11
12     if ($process.ExitCode -ne 0) { `
13         exit $process.ExitCode ; `
14     } `
15     # commenting out the following remove to see if it sets up
16     Remove-Item -Force C:/setup/dotnet-hosting-2.1.3-win.exe
17     ## end install
18     # from https://forums.docker.com/t/ms-access-db-microsoft-jet-oledb-4-0-issue/50541
19
20 ## Install the OLE DB
21 SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop';"]
22 RUN Invoke-WebRequest 'https://download.microsoft.com/download/4/3/9/4393c9ac-e69e-458d-9f6d-2fe191c51469/Jet40SP8_9xNT
23
24 ## end install
25
26 RUN c:\setup\Jet40SP8_9xNT.exe /Q
27
28 SHELL ["powershell"]
29 RUN Remove-Website -Name 'Default Web Site'; `
30     New-Item -Path 'C:\web-app' -Type Directory; `
31     New-Website -Name 'web-app' -PhysicalPath 'C:\web-app' -Port 80 -Force
32 EXPOSE 80
33 RUN Set-ItemProperty -Path 'HKLM:\SYSTEM\CurrentControlSet\Services\Dnscache\Parameters' `
34     -Name ServerPriorityTimeLimit -Value 0 -Type DWord
35 #COPY ProductLaunch.Web /web-app
36 # The final instruction copies the site you published earlier into the container.
37 COPY /bin/Release/PublishOutput/ /web-app
38
39 # to test docker run <imagename> -d

```

The full capabilities and syntax of a Docker file are, as they say, beyond the scope of this article. Instead, here are a couple of notes:

- `escape=`` means we are using the ``` character as a line-continuation or escape character. Docker uses the `/` character as escape by default, so every slash used as a path name would have to be escaped (ie `/bin/Release` would have to be `//bin//Release`). Instead we use ``` so we can specify `/bin/Release`. Confused yet?
- Notice we can invoke PowerShell and run PS commands. This is pretty useful. When you see the file names and paths, bear in mind those refer to the container's file system, not the physical hard drive on your hosting machine.
- Dockerfile commands are shown in UPPERCASE by convention. Notice we only use a few: FROM, SHELL, RUN, EXPOSE, COPY. Most of these are self-explanatory; "EXPOSE 80" opens Port 80 for our HTTP use with our app. You

could expose any port you want--again, you are opening a port on the container, not on your host device.

Where does my app go?

Although it can be done, we're not using Docker to compile and build our application. Normally that is done on a build server after a commit and successful test run. Then the resulting app--following a successful build--is published to a hosting site. We're going to follow a similar workflow: the app is in Visual Studio, where I will build it on my laptop. But at the end of the build, let's have Visual Studio publish the app to a specific location, in this case [my working directory]\bin\release\publishoutput. That way the final step in our Dockerfile can copy all the contents of that directory to the /web-app directory *in the container*.

Next: Build and run

[Part 3 of this series](#) will show how we can build the container and then run the app locally.

VB6 to Docker Part 3

(Note: this is the part 3 in a four part series--if you're just starting here I would urge you to go back to [Part 1 and 2](#) and read them first.

Ok, let's recap briefly.

We know what Docker is: a container engine. And we know containers are a super-lightweight, self-contained, and portable way to run apps. And we know we tell Docker what to do with a dockerfile.

Building our image

Notice I didn't write "building our app." Although you can run a compiler--and thus a build step--inside a container, it's not really a normal approach. As I said earlier, the usual workflow is to automate the build and release cycle on a dedicated build server (a key part of CI/CD). The "build" we're doing here is building an image of our app that can run inside a Docker container.

Docker has a command line interface (CLI) available from PowerShell. Once you've ensured that Docker for Windows is actually running (check the system tray), you can start a PS session and start typing Docker commands. You can always type `Docker --help` and get a list of all the available commands, and for each command you can get a little more detailed help, like with `Docker build --help` to get help on the build command. But for more indepth information you'll have to go to the official documentation, or get a copy of Elton Stoneman's excellent book *Docker on Windows: From 101 to production with Docker on Windows*, available from the [publisher](#) or [Amazon.com](https://www.amazon.com).

Each build fetches the base images you have specified in the dockerfile, and if necessary unpacks and runs installers specified in the base image. A Docker image is typically about 10GB, so you can spend quite a bit of time downloading the initial parts for your first build. However, when you have gone through that once you can modify your build options and even your dockerfile without having to download the *same pieces again*. Of course, if you change your base image, you may find yourself waiting for large downloads all over again.

Let's build our image for SKS. First, we have to set the current path in our PS instance to wherever the dockerfile for this app is. In this case it's in [appname]\upgraded. The command we will use is

```
docker build -t web-app:latest .
```

The `-t` switch tells Docker that we want to use a name for this image, basically a name for the image we can refer to. The format is `name:tag`, where the tag is optional. In this case the image name (tag) is `web-app`, although it could be anything we want it to be. The `:latest` tag is optional and used to clarify which version of the app is in the container (consider things like "stable," "nightly," or "bug-infested.") Finally, the `."` at the end tells Docker to use the current directory as the context for the image.

When we run the build command, Docker processes all the steps in the dockerfile, starting with the FROM and ending with the COPY. As you'll see from the output, it lets you know how the progress is going.

```

PS D:\demo\SKS_Web_5_0_2\Upgraded> docker build -t web-app:latest .
Sending build context to Docker daemon 356.8MB
Step 1/11 : FROM microsoft/aspnet:windowsservercore-10.0.14393.693
--> e761eca2f8df
Step 2/11 : SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop';"]
--> Using cache
--> e64e6d65cbc7
Step 3/11 : RUN New-Item -Type Directory C:\Setup | Out-Null ; Invoke-WebRequest 'https://download.microsoft.com/download/6/E/B/6EBD972D-2E2F-41EB-9668-F73F5FDDC09C/dotnet-hosting-2.1.3-win.exe' -UserAgent '' -OutFile C:/setup/dotnet-hosting-2.1.3-win.exe ; $process = start-process -Filepath C:/setup/dotnet-hosting-2.1.3-win.exe -ArgumentList @('/install', '/q', '/norestart', 'OPT_NO_RUNTIME=1', 'OPT_NO_SHAREDFX=1') -Wait -PassThru ; if ($process.ExitCode -ne 0) { exit $process.ExitCode ; } Remove-Item -Force C:/setup/dotnet-hosting-2.1.3-win.exe
--> Using cache
--> d9308546c74b
Step 4/11 : SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop';"]
--> Using cache
--> 721395a50a1f
Step 5/11 : RUN Invoke-WebRequest 'https://download.microsoft.com/download/4/3/9/4393c9ac-e69e-458d-9f6d-2fe191c51469/Jet40SP8_9xNT.exe' -UserAgent '' -OutFile C:/setup/Jet40SP8_9xNT.exe ;
--> Using cache
--> 0a0d53a6d46a
Step 6/11 : RUN c:\setup\Jet40SP8_9xNT.exe /Q
--> Using cache
--> 755106b3b361
Step 7/11 : SHELL ["powershell"]
--> Using cache
--> 29ec44111e25
Step 8/11 : RUN Remove-Website -Name 'Default Web Site'; New-Item -Path 'C:\web-app' -Type Directory; New-Website -Name 'web-app' -PhysicalPath 'C:\web-app' -Port 80 -Force
--> Using cache
--> c6f2f8131cad
Step 9/11 : EXPOSE 80
--> Using cache
--> 9ff16fb2dca8
Step 10/11 : RUN Set-ItemProperty -Path 'HKLM:\SYSTEM\CurrentControlSet\Services\Dnscache\Parameters' -Name ServerPriorityTimeLimit -Value 0 -Type DWord
--> Using cache
--> 8b4c1a855f32
Step 11/11 : COPY /bin/Release/PublishOutput/ /web-app
--> Using cache
--> 725372cde01e
Successfully built 725372cde01e
Successfully tagged web-app:latest
PS D:\demo\SKS_Web_5_0_2\Upgraded>

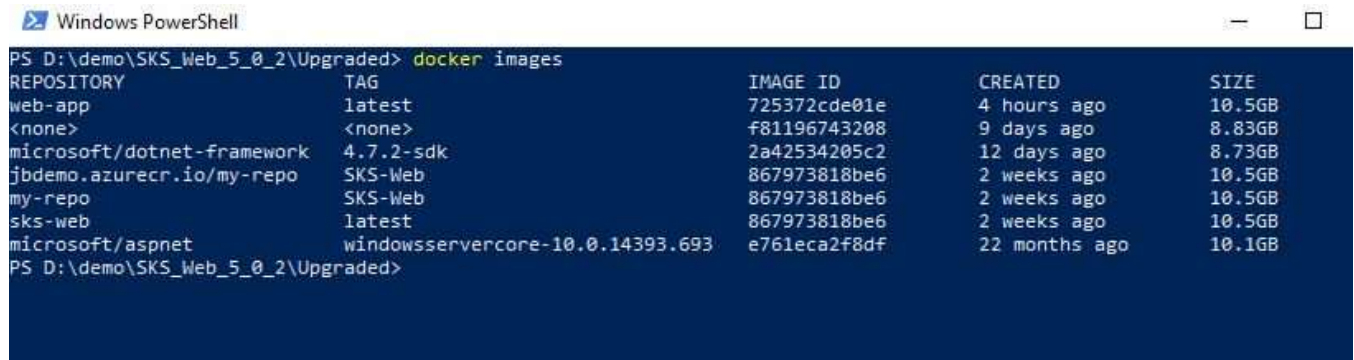
```

This build was fast, but the first time it was **slow**, because downloading and installing the windowsservercore image takes awhile. Take a nap, practice juggling, or just do something else. It will be a minute.

If the build doesn't work, check the steps to see where it failed, then go fix your dockerfile and try again.

Checking and running our image

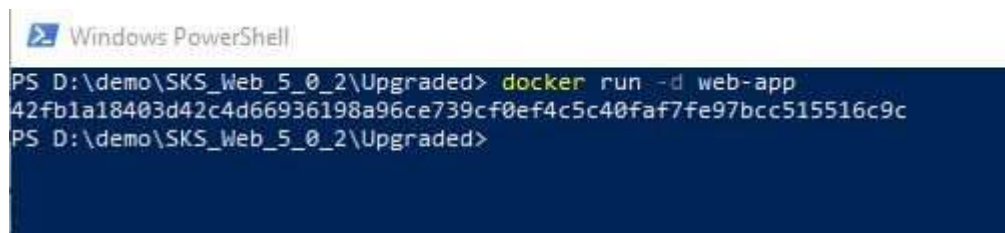
We can make sure the build was good by running the Docker images command:



```
Windows PowerShell
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker images
REPOSITORY          TAG                IMAGE ID           CREATED            SIZE
web-app             latest            725372cde01e      4 hours ago       10.5GB
<none>             <none>           f81196743208      9 days ago        8.83GB
microsoft/dotnet-framework  4.7.2-sdk        2a42534205c2      12 days ago       8.73GB
jbdemo.azurecr.io/my-repo  SKS-Web          867973818be6      2 weeks ago       10.5GB
my-repo             SKS-Web          867973818be6      2 weeks ago       10.5GB
sks-web             latest           867973818be6      2 weeks ago       10.5GB
microsoft/aspnet     windowsservercore-10.0.14393.693  e761eca2f8df      22 months ago     10.1GB
PS D:\demo\SKS_Web_5_0_2\Upgraded>
```

Notice the first line--this is the image we just built. Ignore the rest--they are other images I have created in the past. Notice that each one is about 10GB of disk space--they will stick around on your hard drive until you remove them with the docker system `prunecommand` or the `docker rmi [image]` command.

So we have our image, so what? Well, now we run it, a step which will cause Docker to create a new container and execute the image within the container. Let's do it:



```
Windows PowerShell
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker run -d web-app
42fb1a18403d42c4d66936198a96ce739cf0ef4c5c40faf7fe97bcc515516c9c
PS D:\demo\SKS_Web_5_0_2\Upgraded>
```

Hmm, that was kind of anticlimatic. The relevant parts of the command are the `-d` switch (which says to run the container in the background and print out the container ID) and the image name (`7253`). If you look at the screenshot from the `docker images` command, above, you'll see the image ID is `725372cde01e`. We can usually get docker to act on an image by specifying just enough of that value to be unique. In this case "`72`" threw an error, but "`7253`" was enough to let Docker know what image I wanted to run. Go figure.

We can now verify that the container is alive and running with the `docker ps` command:

```
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
gracious_mendel    cc/f44f736898     7253               "C:\\ServiceMonitor.e..." 2 minutes ago      Up About a minute  80/tcp
PS D:\demo\SKS_Web_5_0_2\Upgraded>
```

This command lists all running containers. If you've been messing around with Docker for awhile, try `docker ps -a` which will list ALL containers:

```
Windows PowerShell
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
42fhlai18483d     web-app            "C:\\ServiceMonitor.e..." 46 hours ago       Exited (3221225786) 21 hours ago      unruflled_hamilton
6b3f96d49c64     sks-web           "C:\\ServiceMonitor.e..." 5 days ago         Exited (3221225786) 1 days ago       vibrant_johnson
6b2ab8f8329e     sks-web           "C:\\ServiceMonitor.e..." 8 days ago         Exited (3221225786) 8 days ago       adoring_lewin
bc1c37be9e05     86                "C:\\ServiceMonitor.e..." 8 days ago         Exited (0) 8 days ago    unruflled_loveleace
1ef145819c89     f81196743288     "powershell 'Remove-..." 11 days ago        Exited (1) 11 days ago    fervent_swirles
f71e32fc4386     f8                "id"                11 days ago        Created
1b36486417ae     f81196743288     "powershell 'Remove-..." 11 days ago        Exited (1) 11 days ago    modest_kirch
f8b646e719f5     f8                "c:\\windows\\system32..." 11 days ago        Exited (0) 11 days ago    thirsty_shaw
39d24a6623c7     f8                "c:\\windows\\system32..." 11 days ago        Exited (0) 11 days ago    suspicious_yonath
c3e82ae6970b     f81196743288     "powershell 'Remove-..." 11 days ago        Exited (1) 11 days ago    xenodochial_shirley
2e494ac5351f     sks-web           "C:\\ServiceMonitor.e..." 2 weeks ago        Exited (3221225786) 2 weeks ago    friendly_jackson
bfe5d14d8295     sks-web           "C:\\ServiceMonitor.e..." 2 weeks ago        Exited (0) 2 weeks ago    brave_saha
PS D:\demo\SKS_Web_5_0_2\Upgraded>
```

What's useful here is the the column PORTS (80/tcp) and NAMES (gracious_mendel). We can access our container via good old TCP port 80, and get information about it using the name gracious_mendel (I'm delighted by the algorithm for naming containers, as it uses a random adjective and the name of a famous scientist. This one immediately took me back to a genetics class at university, putting fruit fly generations to sleep with chloroform to look at them under a microscope.)

Small detour

I'm writing this paragraph a day after writing the above one. So look what happens:

```
docker inspect gracious_mendel
```



```

PS D:\demo\SKS_Web_5_0_2\Upgraded> docker inspect gracious_mendel
[
  {
    "Id": "ccf44f73689889c101f8e8a413b7d914067967ed8ee9d103997946e120ccd1f3",
    "Created": "2018-11-26T21:33:30.005584Z",
    "Path": "C:\\ServiceMonitor.exe",
    "Args": [
      "w3svc"
    ],
    "State": {
      "Status": "exited",
      "Running": false,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false
    }
  }
]

```

Notice under State that Running is "false" and "status" is "exited." Last night I shut down the container with a docker stop command.

If I try to start it, I can't--at least not with the old name:

```

PS D:\demo\SKS_Web_5_0_2\Upgraded> docker run -d gracious mendel
Unable to find image 'gracious:latest' locally
C:\Program Files\Docker\Docker\Resources\bin\docker.exe: Error response from daemon: pull access denied for gracious, repository does not exist or may require 'docker login'.
See 'C:\Program Files\Docker\Docker\Resources\bin\docker.exe run --help'.
PS D:\demo\SKS_Web_5_0_2\Upgraded>

```

The name gracious_mendel applied to a running instance of the container. Once I stop the container, that name won't work anymore. I have to use the image name to start it, and I will get a new instance name (in this case "sleepy_bartik"). Goodbye Gregor:

```

PS D:\demo\SKS_Web_5_0_2\Upgraded> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
166b7e8dd65        7253               "C:\\ServiceMonitor.e" About a minute ago  Up About a minute  80/tcp              sleepy_bartik
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
166b7e8dd65        7253               "C:\\ServiceMonitor.e" About a minute ago  Up About a minute  80/tcp              sleepy_bartik
ccf44f736898      7253               "C:\\ServiceMonitor.e" 2 days ago         Exited (3221225786) 25 hours ago         gracious_mendel
42fb1a18483d      web-rog            "C:\\ServiceMonitor.e" 2 days ago         Exited (4204967295) 2 days ago         unruffled_hamilton

```

Let's inspect this instance:

docker inspect sleepy_bartik

Windows PowerShell

```
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker inspect sleepy_bartik
[
  {
    "Id": "166b7e8ddd65c11dff3bba25e67d848685098ffec05bdcfba9c453611655aa5c",
    "Created": "2018-11-28T22:33:27.6121203Z",
    "Path": "C:\\ServiceMonitor.exe",
    "Args": [
      "w3svc"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 2032,
```

You can see that this one may be sleepy but it's also running. The docker inspect command provides a ton of information from some XML data. [You can use go templates](#) to do interesting things with it. What I want at the minute is the public IP address to access it via TCP port 80. Here's a go template to print the IP address:

```
docker inspect -f '{{ .NetworkSettings.Networks }}' sleepy_bartik
```

and it will print

```
172.19.209.167
```

```
]
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker inspect -f '{{ range .NetworkSettings.Networks }} {{ .IPAddress }}{{end}}'
sleepy_bartik
172.19.209.167
PS D:\demo\SKS_Web_5_0_2\Upgraded>
```

I want some fish

Hey, I have an app for that. Let's type our IP address into a browser:

Angular

172.19.209.167/

File ▾ Orders ▾ Inventory ▾ Maintenance ▾ Help ▾

Create Order

Search

Company

First Name Last Name

Text	Customer	Company	First Name	Last Name	City	State
2	Wappec	Ronald J.	Ratzlaff	Lexington	NC	Country/R...
11	World of Fun	Jessica L.	Sexton	Scotts Hill	NC	Country/R...

Customer

Company Contact

Required by Promised by

Quantity	Code	Product	UnitPrice	Price	Existence	Ordered	Quantity...
	SH22	Shark	2.25		0	14	2gr

Quantity

Sales Tax

Freight

Total

Total Tax

Sub Total

Ok, it works. Here's our Salmon King Seafood order form running as a web app in localhost, talking to the IP address of our container.

Next: [let's get it working on Azure](#).

VB6 to Docker Part 4

Almost done.

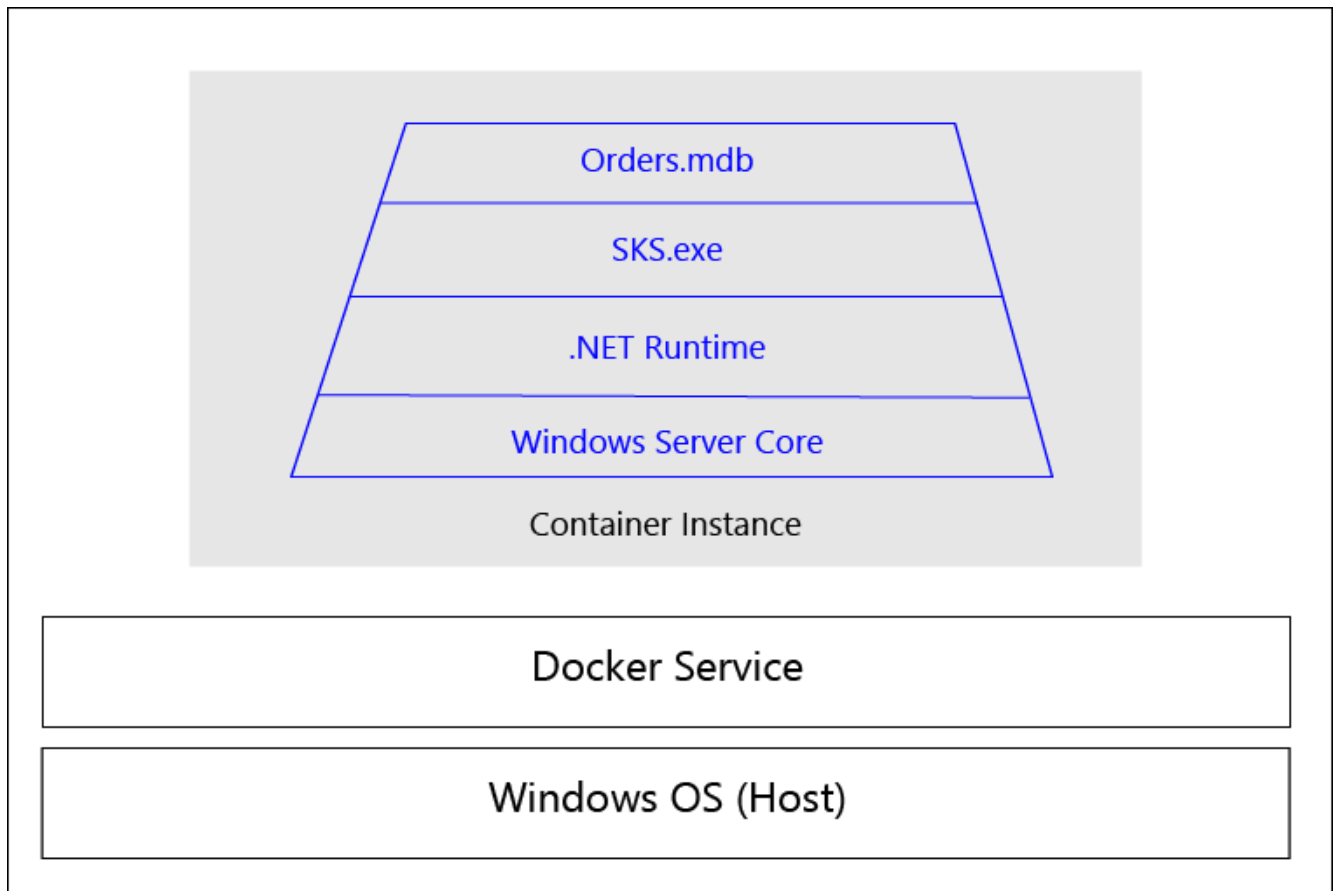
If you've been reading along, you know we got Docker for Windows running, created a dockerfile in Visual Studio, and published our app to a directory where Docker could find it. Running our web app in the container allows us to access it via localhost:80 and it runs fine.

If you're starting here, you might want to go back to [Part 1](#) and read from the beginning.

Now that our app is running in a local container, we can move that container image anywhere the Docker for Windows environment is available and run it there. The beauty of that--and I can't stress this enough--is that the image, being completely self-contained, will run anywhere the service is available. So there's basically no way to hose it when you put it into production, or move it from one deployment environment to another.

What's in the container?

As a refresher, let's look at the contents of our newly created container.



The container instance runs inside the Docker service, which in turn runs on--in this particular case--a Windows OS. Notice I didn't write Windows Server or Windows 10. Any Windows OS that the Docker service can run on can also run this container. In the case of Docker for Windows Community Edition, it has to run on Windows 10 64-bit Pro, Enterprise, or Education (not Home). But the container can run on any supported Windows OS (as we'll see when we push it to Azure).

Why Azure?

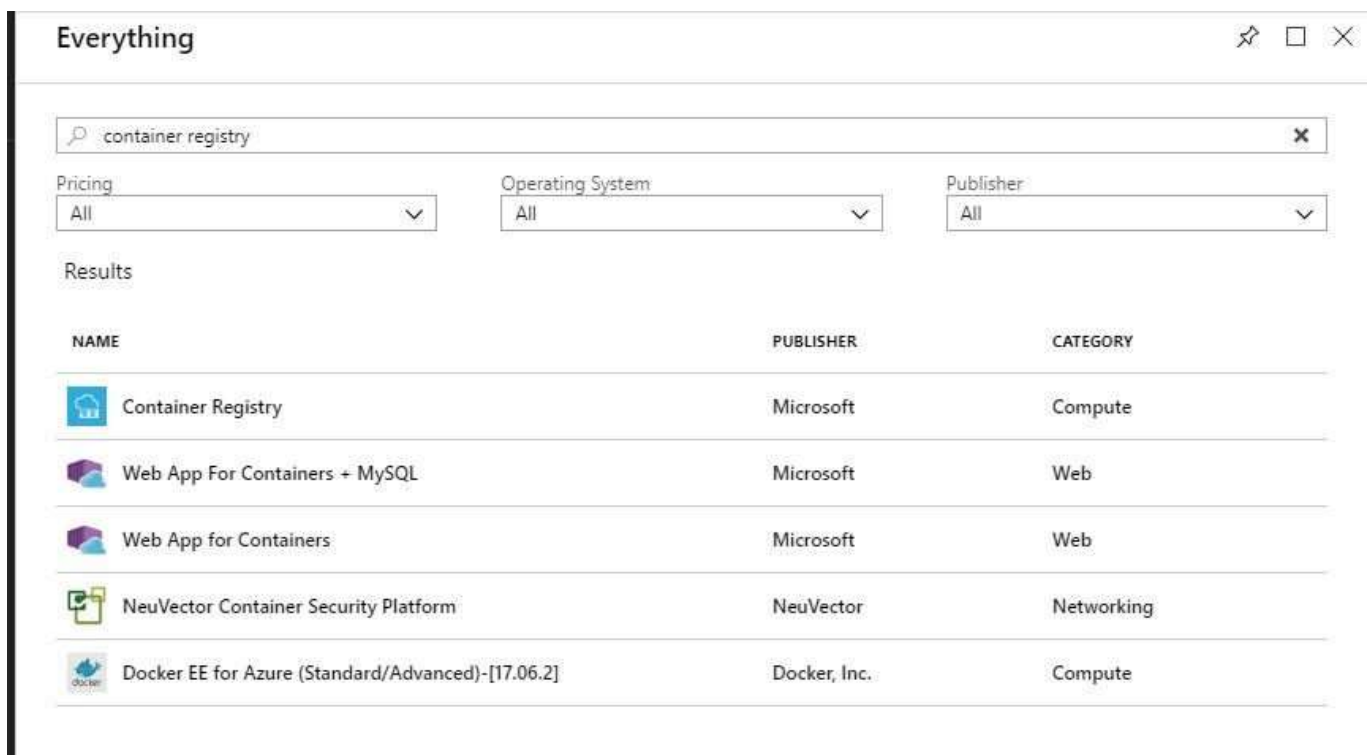
No particular reason, except it's a little easier/simpler to get this lashup running on Azure than it is on AWS, and I wanted to close this thread with moving the app to the cloud. In future if there is interest I'll do one on AWS. But this time is Azure.

Azure gives you--in its typically mind-numbingly confusing way--two different ways to run a Docker container. There could be 20, who knows? The one I found first is to use a container registry, but you can also run container instances. At least I think so. Who knows? So for this tutorial, we will use container registries. Supposedly this is designed

to support swarms, which is a plurality of Docker containers (like a *murder of crows* or a *parliament of owls* I suppose). As mentioned earlier, one of the benefits of containers over VMs is how much faster you can spool one up, so having a swarm of identical containers lets you exercise some of that vaunted cloud elasticity in a more responsive way.

Creating a registry

First you need an Azure account--and there are an equally mind-numbing number of choices in that regard. You're on your own, Bucky. Once you are logged in, either go to your resource group or create a resource group (a billing unit) and click Add. So many choices! Like when I go to buy earrings for my wife--they show me like 500 pairs and eventually I just close my eyes and point. So in the search box, type "container registry" and you find it:



When you click on it, you get a nice explanation from Microsoft about what this is, and a button to create one. Here's what they say:

Azure Container Registry is a private registry for hosting container images. Using the Azure Container Registry, you can store Docker-formatted images for all types of container deployments. Azure Container Registry integrates well with orchestrators

hosted in Azure Container Service, including Docker Swarm, DC/OS, and Kubernetes. Users can benefit from using familiar tooling capable of working with the open source Docker Registry v2.

Use Azure Container Registry to:

- Store and manage container images across all types of Azure deployments*
- Use familiar, open-source Docker command line interface (CLI) tools*
- Keep container images near deployments to reduce latency and costs*
- Simplify registry access management with Azure Active Directory*
- Maintain Windows and Linux container images in a single Docker registry*

Clicking "Create" takes you to a form where you have to specify the name, resource group, and some other stuff. For SKU select "Standard."

Create container registry □ ×

* Registry name
Enter the name
.azurecr.io

* Subscription
Presales

* Resource group
Select existing...
Create new

* Location
West US

* Admin user ⓘ
Enable Disable

* SKU ⓘ
Standard

Notice in *tiny little letters* under the Registry Name field is .azurecr.io. Your container registry will get assigned a login server named <registry name>.azurecr.io. Since I created a registry called JBDemo (such hubris), my login server name is jbdemo.azurecr.io. You'll need this name later to push the container up to Azure.

Here's what you should see next:

JBDemo
Container registry

Search (Ctrl+F)

→ Move Delete Update

Overview

- Activity log
- Access control (IAM)
- Tags
- Quick start
- Events

Settings

- Access keys
- Locks
- Automation script

Services

- Repositories

Resource group (change)
Marketing

Location
West US

Subscription (change)
Prasales

Subscription ID
3c6a49f7-84f6-4aaf-9d7e-01250b70e3f0

Login server
jbdemo.azurecr.io

Creation date
11/12/2018, 2:07 PM PST

SKU
Standard

Provisioning state
Succeeded

Registry quota usage

Used
0.3 GiB

Available in SKU
99.7 GiB

100 GiB
SIZE QUOTA

ACR Tasks

Build, Run, Push and Patch containers in Azure with ACR Tasks. Tasks supports Windows, Linux and ARM with QEMU.

[Learn more](#)

If you don't see this, just go to your resource group (mine is called "Marketing") and you'll see all your resources:

The screenshot shows the Azure portal interface for a resource group named 'Marketing'. The left-hand navigation pane includes sections for Overview, Activity log, Access control (IAM), Tags, Events, Settings (Quickstart, Deployments, Policies, Properties, Locks, Automation script), Cost management (Cost analysis, Budgets, Advisor recommendations), and Monitoring (Insights (preview), Alerts, Metrics, Diagnostic settings). The main content area displays a list of 14 items with the following columns: NAME and TYPE. The items are as follows:

NAME	TYPE
az-marketing-1	Virtual machine
az-marketing-1	Network security group
az-marketing-1	Public IP address
az-marketing-1_0bdf522295b146e189646498245b7ae2	Disk
az-marketing-15	Network interface
ChefServer	Network security group
ChefServer	Public IP address
chefserver509	Network interface
JBDemo	Container registry
jbsks	Container instances
Marketing	Virtual network
marketing9025	Storage account
sksazure	Container instances
sksweb	Container instances

You can see there is one container registry (JBDemo) and three container instances (jbsks, sksazure, and sksweb). Click on any of these to open it up--I'm going to open my container registry. And you'll notice there are not containers, nor is there any obvious way to upload a container.

Yeah, that stalled me for awhile.

Pushing the Docker container to Azure

Ok, now we have to go back to our PowerShell window where we ran our container. At this point I don't really need this container running on my local machine anymore, so I can stop it with this command:

```
docker container stop gracious_mendel
```

Docker will respond with the name of the container ("gracious_mendel") indicating it is stopped. I can confirm that by running the Docker ps command, which will show no running containers:

```
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker container stop gracious_mendel
gracious_mendel
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
```

Now we are ready to push this container image up to Azure. But how will Docker know which image to push? The answer is the tag command:

```
docker tag web-app jbdemo.azurecr.io/repos/sks:latest
```

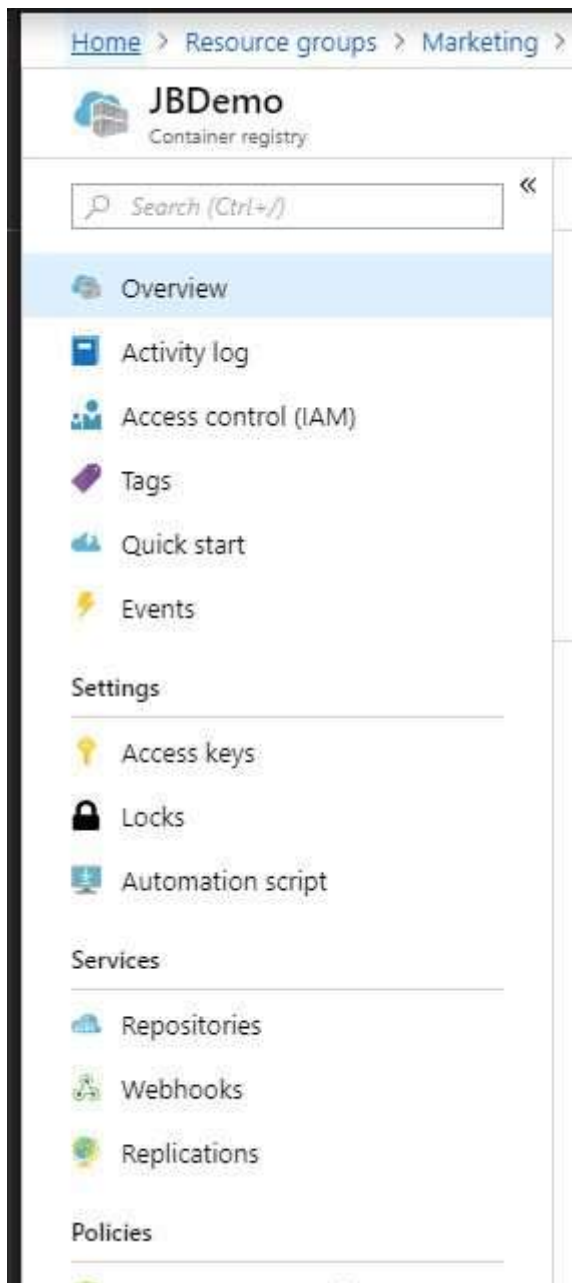
```
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
web-app             latest             725372cde01e       29 hours ago       10.5GB
<none>             <none>            f81196743208       11 days ago        8.83GB
microsoft/dotnet-framework  4.7.2-sdk         2a42534205c2       13 days ago        8.73GB
my-repo             SKS-Web            867973818be6       2 weeks ago        10.5GB
sks-web             latest             867973818be6       2 weeks ago        10.5GB
jbdemo.azurecr.io/my-repo  SKS-Web            867973818be6       2 weeks ago        10.5GB
microsoft/aspnet    windowsservercore-10.0.14393.693  e761eca2f8df       22 months ago      10.1GB
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker tag web-app jbdemo.azurecr.io/repos/sks:latest
PS D:\demo\SKS_Web_5_0_2\Upgraded>
```

All this command does is take the image called web-app (see the output of the Docker images command above) and associate it with a name on my container registry, in the form of <server name><repo name><container name:tag>.

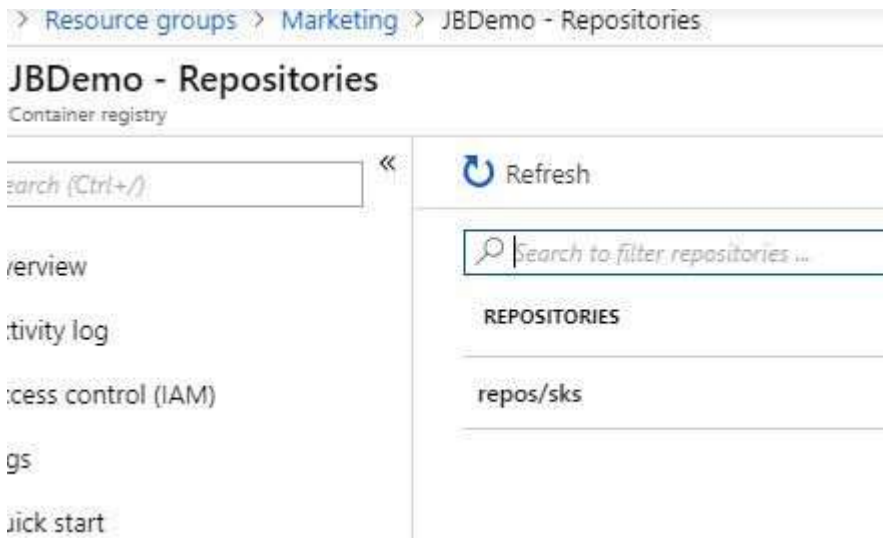
Ok, let's have Docker copy this image to our container registry:

```
sks-web             latest             867973818be6       2 weeks ago        10.5GB
jbdemo.azurecr.io/my-repo  SKS-Web            867973818be6       2 weeks ago        10.5GB
microsoft/aspnet    windowsservercore-10.0.14393.693  e761eca2f8df       22 months ago      10.1GB
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker tag web-app jbdemo.azurecr.io/repos/sks:latest
PS D:\demo\SKS_Web_5_0_2\Upgraded> docker push jbdemo.azurecr.io/repos/sks:latest
The push refers to repository [jbdemo.azurecr.io/repos/sks]
6e56d63087be: Pushing [=====] 70.64MB/74.81MB
ce332d721395: Pushed
307cd6d5ee86: Pushed
930504ceec4d: Pushed
7afe6405e7e3: Pushed
e4e258563d8c: Pushed
5181a9a07a0c: Pushed
374b38af54d9: Pushed
34feacfbb084: Pushing [=====] 67.81MB
ab4fa5a78263: Pushed
ce74595d58f5: Pushing [=====] 55.73MB/250.2MB
5b4aaace84103: Pushed
1f2f3eb32edc: Pushing [=====] 192kB
0451551dda21: Pushing [=>] 7.027MB/260.3MB
c28d44287ce5: Waiting
f358be10862c: Waiting
```

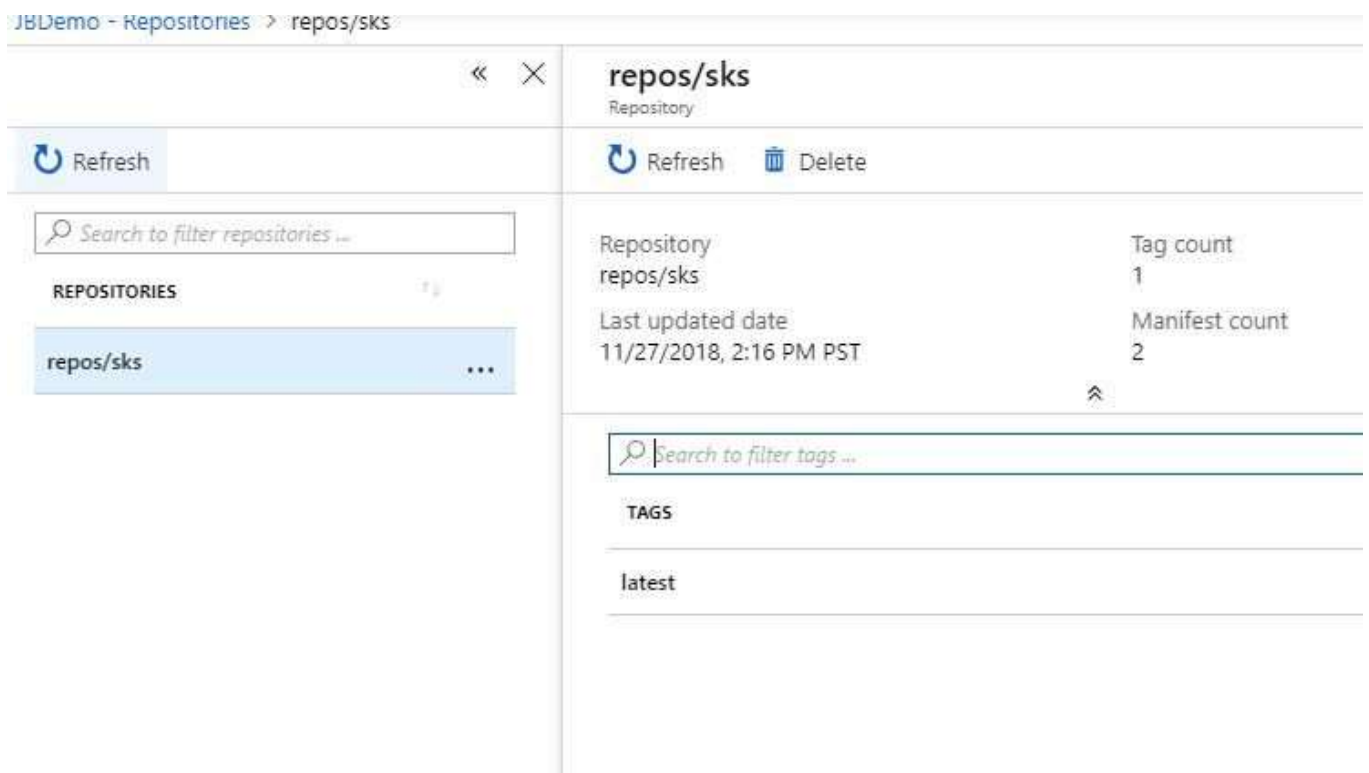
Assuming no errors, you should see something like the screenshot above. When it's finished, go back to your Azure container repository and click "Repositories" from the side bar menu:



There are a number of options here like access control or webhooks. I'll leave that for you to experiment with. For now, we want to look at our repo--no worry that you didn't create a repository in Azure, the push command created one for you:



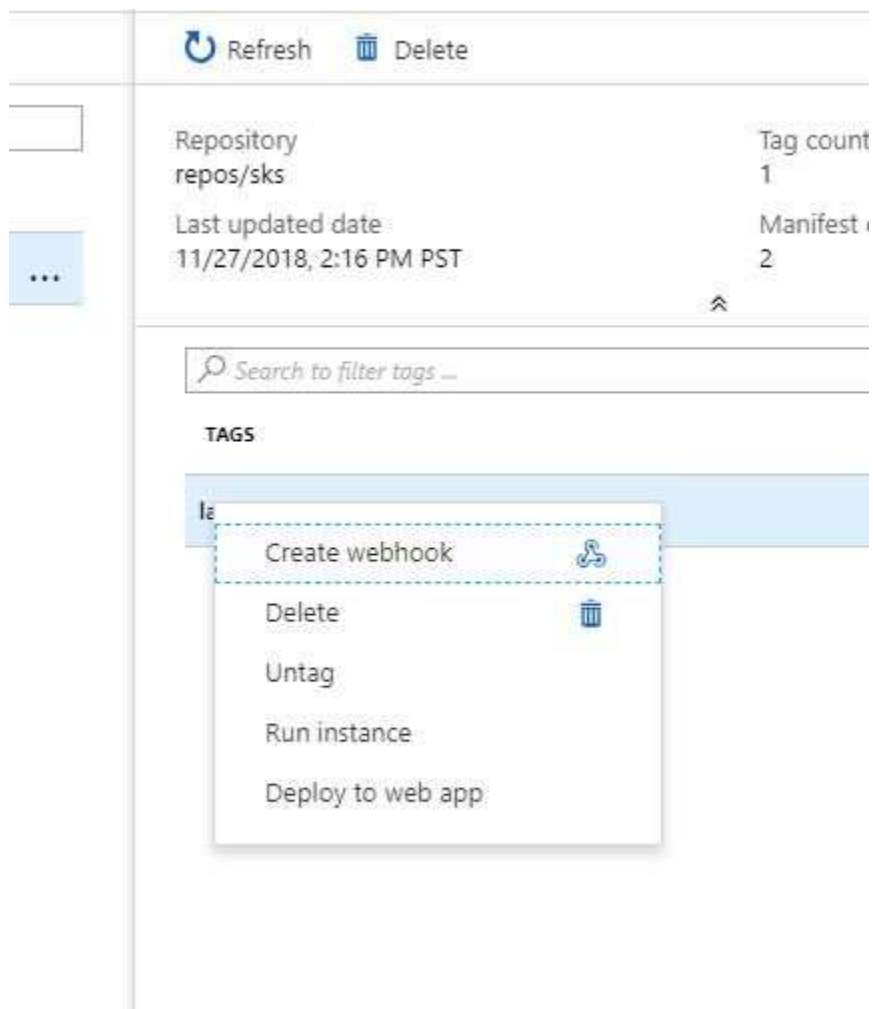
There's nothing magical about the name repos/sks: it's just a name. Let's click on it:



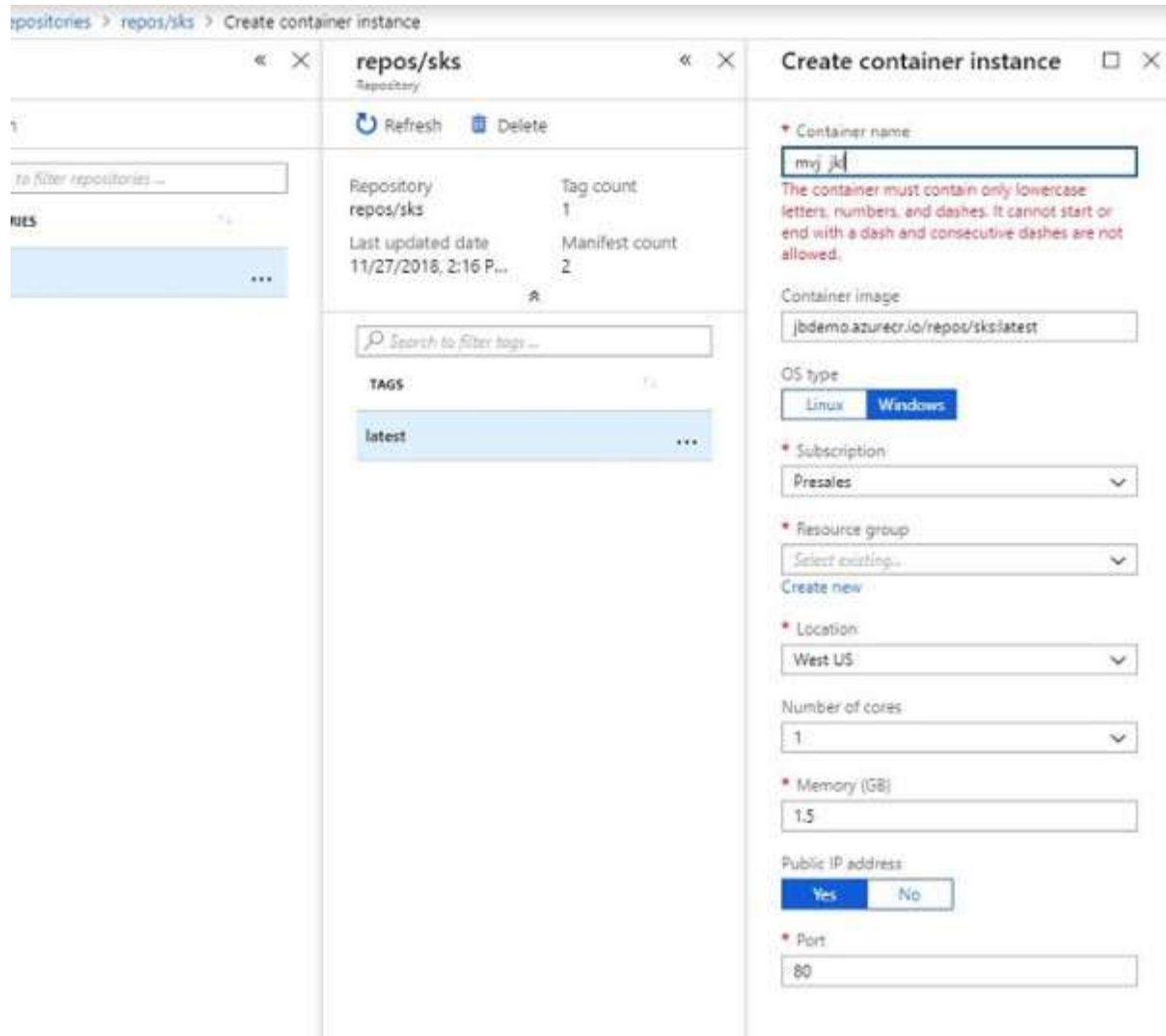
And here we'll see all the tags we've used for this repo. Could be "nightly" or "latest" or "foobar", doesn't matter.

Running the Docker container in Azure

All we have to do is right click on the "latest" tag (or left click on the three dots on the right) and select "Run instance."

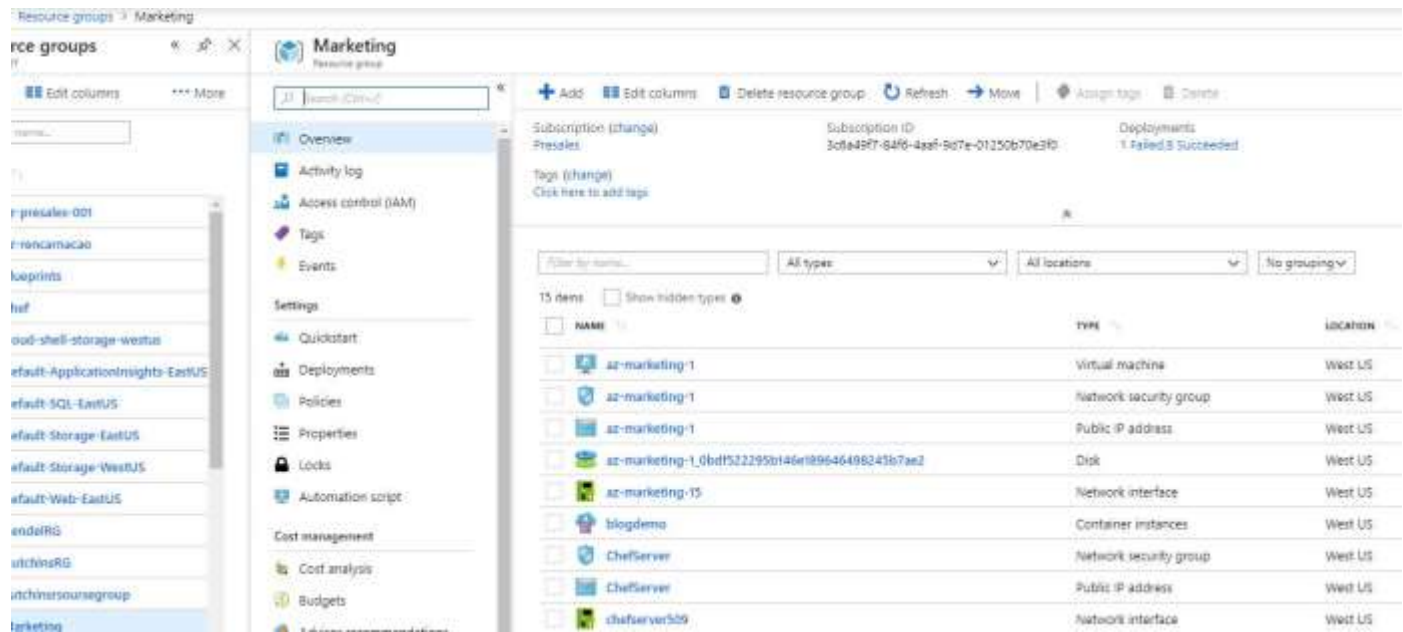


That opens a config screen:

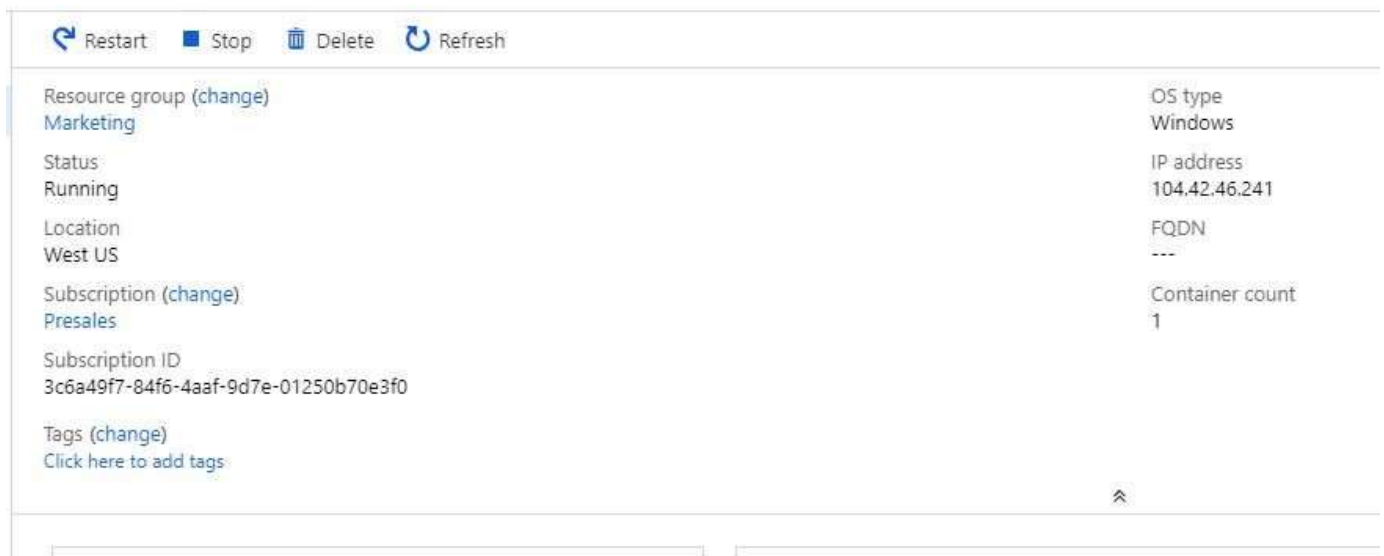


Notice the whinging about my garbage name. I did that so you could see the rules: only lowercase letters, no spaces. Seems pretty restrictive but there it is. Be sure to select Windows as the OS type and Yes for Public IP address. For some reason you have to specify the Resource Group again, so pick the one you started with (in my case it's Marketing).

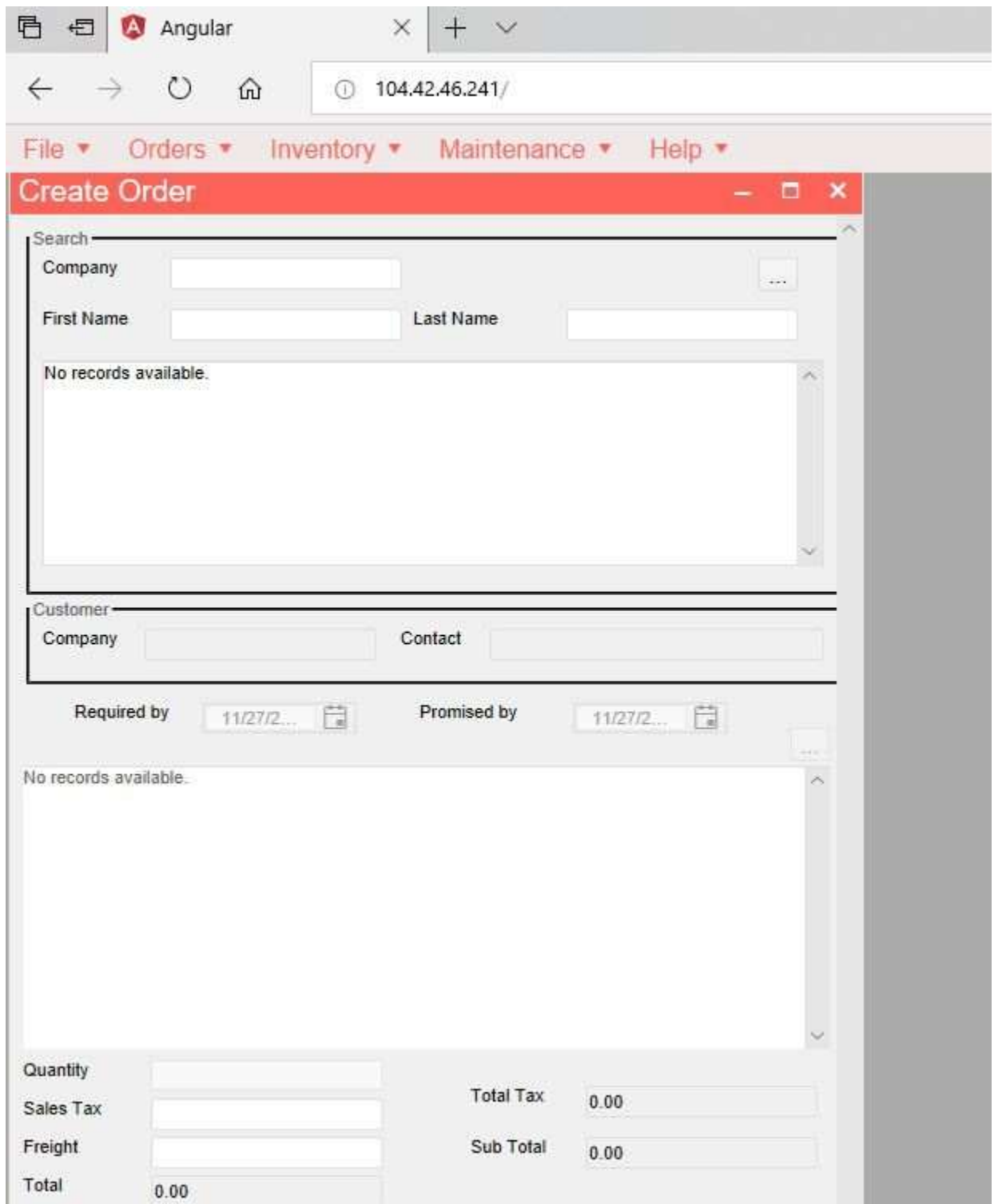
Now go get a cup of coffee, or your beverage of choice. This will take a few minutes. Eventually in the Alarm area (the bell icon top right) you will get a notification that the deployment is successful. If you then go to the instance--go back to your resource group and find the instance, like below (it's called blogdemo):



And there it is. Let's click on it to get the public IP address:



In addition you have a control to stop, delete, or restart the container. Let's go to a browser and connect to this IP address:



and there's our app. Since I literally never touched the source code from my laptop to Azure, I don't need any tests to know this is a good build--well, assuming the version on my laptop was good, this one will be identical--because it's bit for bit identical. And that's the coolness of containers.

Recap

We began with something horrifying yet common: a VB6 desktop app still in use. We made it a modern web app (using [WebMAP](#)) with Angular 6 for the client and ASP.NET Core on the server side, with all business logic intact. Then we put that whole enchilada into a Docker container, ran it locally, then pushed it to Azure and ran it there.

The reasons for moving off the desktop to the web just get stronger. And CI/CD is-- when you really get it--one of the best. But you can't do CI/CD with a desktop app, not really. And containers--like Docker--make CI/CD so much easier, safer, and more reliable. But you have to have a web app. So [take another look at WebMAP](#), because every day it makes it a little easier to get off the desktop and into the 21st century.

Topics: [Docker](#), [containers](#), [webapps](#)