# **Additional Demos**

# **Startup Times**

-One of the most basic values Docker is how fast container start.

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| **Docker ps** | We can see we have nothing running. There’s nothing warmed up |
| **docker run alpine echo hello** | We’re going to simple say Hello from a container.  Since we didn’t have the Alpine image downloaded, docker run will first pull alpine from the public docker hub.  Once it’s cached, it will simply runt the command we’ve passed to it. |
| **docker run alpine echo hello** | If we run it again, we can see it’s much faster as Docker has already cached the image |
| **docker rmi alpine -f** | Let’s do that again. This time, will measure the time |
| **measure-command {docker run alpine echo hello}** | <4 seconds to download and run the image |
| **measure-command {docker run alpine echo hello}** | Less than a second to run the image, once cached.  Imagine trying to accomplish that with a VM Image… |

# **Batch Jobs, containing dependencies**

You need to run some process to do some work. That work has some dependencies to run. It may be a connection to a database, a public service, an image manipulation or map routing library. It may be DNA modeling algorithms. Do you want to install all those dependencies to run locally? Can you install them?

Let’s build a container that calculates Magic 8 Ball Answers.

To give a response, we need the list of possible questions. I don’t know what they are. Do you?

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| **docker run stevelasker/magic8ballbatchsample "Will my demos work?"** | We can see we have nothing running. There’s nothing warmed up |

# Building the release image

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| FROM microsoft/aspnetcore:1.0.1  WORKDIR /app  EXPOSE 80  COPY . .  ENTRYPOINT ["dotnet", "Tradapp.dll"] | Now that we have the output ready to publish, we simply place that output into an ASP.NET Core Optimized image  We add EXPOSE to let docker run know what ports the container is “doing business on” |
| "publishOptions": {  "include": [  "Dockerfile", | The last thing we’ll do is make sure our dockerfile is copied to our published directory |
| **Docker build . -f dockerfile.build -t tradapp:build** | From the root of our source directory, we compile our app in the .build environment |
| **docker run -d --name tradapp-build tradapp:build** | Instance the container |
| **docker cp tradapp-build:app/publish/ publish** | Copy the compiled results to our publish directory |
| **docker rm -f tradapp-build** | Stop the build container |
| **docker build publish -t tradapp:latest** | Build our final image |
| **docker run -d -p 8080:80 tradapp**  **Localhost:8080** | Test our final image |
| **Docker images** | Now we can see a pretty big difference between our :build and :latest image |

# **Building Optimized Images**

If you’ve followed basic docker walkthroughs, you’ve liked found steps like the following. Lets take this journy

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| **Create New Project** | We’ll create a new .NET Core Web app |
| **Add a Dockerfile** FROM microsoft/dotnet  WORKDIR /app  ENV ASPNETCORE\_URLS http://\*:80  EXPOSE 80  COPY . .  RUN dotnet restore  ENTRYPOINT dotnet run | We’ll add an empty dockerfile and the standard content  The ENV is important as we need to tell the web server (Kestrel) to listed to traffic beyond localhost. In a container, localhost will only work on the private network of docker  We expose port 80, on the container. The host port comes in our docker run command  We Copy the source into the container  Run dotnet restore, to restore all our packages  Set the entry point to dotnet run |
| **Docker build . -t tradapp** | From the root of our source, we’ll do a docker build, using the current directory as the “context” .  Context means, this is the directory to send to the host. The copy command copies content from the context directory to the destination specified in the container. The WORK\_DIR or /app  -t means tag the image with a name, so we can find it in our list |
| **Docker run -d -P tradapp** | Lastly, we can simply call run  We pass a few extra parameters:  -d means start the container and detach  -P means takes all the ports EXPOSEd and find an available port on the host and dynamically assign it |
| **Docker ps** | We now run a docker ps to find the port assigned |
| **Localhost:{port}** | If we open the browser to the dynamically assigned port, we can see our app.  Congrats, we have a .NET Core app built and running in a container… |

# **Size and Perf matter**

In a container world, where containers can startup at any time, on any node across the cluster, the speed by which we can pull the image across the network, instance the container and have it start serving requests is the top priority

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| **Docker images** | Looking at the images we just created, let’s see some sizes |
| **Switch to DockerStartTime dotnet run "run -d -p 8080:80 tradapp" "**<http://localhost:8080>**"** | Lets time how fast this image builds and runs  We’ll use this little DockerStartTime applet I created. It simply uses a StopWatch to capture the amount of time docker run takes to start the container. Then, how much additional time for ASP.NET Core to start serving requests |
| **Container Start Minutes : 0**  **Container Start Seconds : 0**  **Container Start Milliseconds : 567**  **Requesting url:** <http://localhost:8080>  **...**  **TotalMinutes : 0**  **TotalSeconds : 15**  **TotalMilliseconds : 433** | Looks like it took 56 milliseconds to start the container  And, 15 seconds to start serving requests.  Much faster than a VM. But, we can do much better |
| **Docker ps**  **Docker exec -it {id} bash** | To get a sense of why it’s so big, we’ll navigate into the container  We first need to get the container id, so we can execute a bash prompt, interactively |
| **cd ~/.nuget/packages/** | We’ll navigate into the directory of our restored packages |
| **find . -type d | sed -e "s/[^-][^\/]\*\// |/g" -e "s/|\([^ ]\)/|-\1/"** | We can display a directory tree to show all the packages. Who wants to type this one in? |
| **Du -h** | Or, we can look at the directories and the total file size |

# **Building optimized images**

To build an optimized image, we need a few things

* The compiled output we want to run
* An optimized image we want to run in.

An optimized image can mean many things. In our case, it means small. It only contains the runtime dependencies we need. We don’t need the SDK, the debugger, Node, bower or other stuff we need to compile the app

We also want everything pre-jitted, or as much as possible

We have a microsoft/aspnetcore image that represents our runtime, optimized image. You can’t build aspnetcore apps with it, but you can run them. The ASP.NET Core nugets are pre-cached in the image, and pre-jitted

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| Update the Dockerfile  FROM microsoft/aspnetcore:1.0.1  WORKDIR /app  COPY . .  ENTRYPOINT dotnet Tradapp.dll | Let’s build an optimized image.  The first thing, is we want to start with the optimized image  Next, we can remove all the build steps. We simply need to copy the contents of the publish folder.  Notice, I’m copying ., which means I’m running in the publish folder. This just minimizes the context of content sent to the host. |
| From the root of your web project **dotnet restore**  **Dotnet publish -c release -o Publish** | On our laptop here, we have everything we need to compile the optimized code.  We’ll restore the packages  And Publish the output. Remember, Publish implicitly does a build. Also, remember, Publish doesn’t delete content first, so you’ll likely want to del \*.\* in the publish directory first. |
| **Explorer .**  Properties on the Publish folder | Let’s explorer what’s created here |
| **Cd Publish**  **docker build . -t myweb:optimized**  unable to prepare context: | We can now build our image, from the published output  Oh, well, maybe not |
| **VS – Edit project.json** "publishOptions": {  "include": [  "Dockerfile", | We need to tell VS and dotnet we want to include the Dockerfile in the published output |
| **Dotnet publish -c release -o Publish** | We’ll republish the app, with the dockerfile |
| **cd Publish**  **docker build . -t tradapp:optimized** | Now, we can build our optimized image |

# **Executing Scripts**

You have some work you need to do. You need to run a series of commands, a script, but the script has dependencies you may not have on your computer, or you may not want on your computer. How would you do that with Docker?

## Getting it to work

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| VS Code | We’ll use VS Code to create a complex script that has all sorts of dependencies. |
| New file: doit.sh | We’ll create a new bash script |
| echo "hello from the inside of this cozy container" | We’ll add some contents. Now, I know what you’re thinking. Wow, I couldn’t run that on my computer. We’re getting started here… |
| docker run -v c:/demos/scratch:/src alpine src/doit.sh | We’re going to run this script in a container, which has the environment we need. We’re going to start with the smallest image available: alpine  Notice we get a very unhappy error:  standard\_init\_linux.go:175: exec user process caused "no such file or directory" |
| docker run -v -it c:/demos/scratch:/src alpine bash | Lets see if our file exists Error response from daemon: oci runtime error: exec: "bash": ex  ecutable file not found in $PATH.  Looks like we can’t even run bash. |
| Docker images | Notice it’s just 4.799mb WOW, but we can’t do much with it |
| docker run -it -v c:/demos/scratch:/src debian bash  ls  cd src/ | We’re going to run this in a Debian image that’s a little bigger but at least has bash.  We can see our src/ directory, and if we navigate into it, we can see our doit.sh script |
| ./doit.sh | Looks like it works. |
| docker run -v c:/demos/scratch:/src debian src/doit.sh | Lets try it again  standard\_init\_linux.go:175: exec user process caused "no such file or directory"  Now that’s not right. We know it’s there  Yup, but this is where we’re crossing streams. Windows, by default, adds CRLF to each line. Linux doesn’t like these. When you run the script directly in the container, it seems to be fine.  When you pipe it through the wormhole, nope.  SO, what to do? |
| Download Dos2Unix | There are likely many ways. Let me show you one interesting utility  I’ve already installed it, and set it in my path |
| Dos2Unix doit.sh | By simply passing doit.sh to the converter, it will remove the carriage returns |
| docker run -v c:/demos/scratch:/src debian src/doit.sh | Wow, that was easy… NOT! ☹ |
| VS Code 🡪 Preferences 🡪 User Settings add  "files.eol": "\n" | We can set VS Code to remove CRs on line endings… |
| New File: doitagain.sh  #!/bin/bash  echo Now, that was easy… | We’ll create a new file and another message |

# **Building .NET Core in a Container**

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| VS: Create a new .NET Core Web App in | Using Visual Studio, we’ll create a new ASP.NET Core Web App |
| **docker run -it -v c:/Demos/scratch:/src microsoft/aspnetcore-build bash** | To build this, in a container, we’ll use the aspnetcore-build image. It has all the dependencies we need to build. |
| **cd src/**  **ls**  **dotnet restore**  **dotnet publish -c release -o publish** | Now that we’re in a container, we can see our source volume mounted in  We’ll restore the nuget packages  And create the published output, using the release configuration  We’re using -o to simplify the path. Otherwise it goes into /src/bin/release/netcoreapp1.0/publish |
| Open Explorer on Windows | Notice the published output. It looks like a lot of files, but the entire content is only 9mb |
| **Exit** | Exit the container |
| **Cd publish**  **Dotnet src.dll** | Let’s test this out  We’ll start the website here from our publish directory |
| **Browse localhost:5000** | Yup, it’s working just fine |
| **dotnet restore**  **dotnet publish -c release -o publish** | Lets automate this |

# **Building .NET Core in a Container with a Dockerfile build script**

Now that we’ve built this in a container, we can see how we can encapsulate the build environment

We can put this in a script to build in the container, or we can use a Dockerfile, which is a build script

|  |  |
| --- | --- |
| FROM microsoft/aspnetcore-build:1.0.1  WORKDIR /app  COPY . .  ENV NUGET\_XMLDOC\_MODE Skip  RUN dotnet restore  RUN dotnet publish -c release -o publish | We’re going to convert our script to a Dockerfile.build  By doing this, we can benefit from docker image layer caching  We’ll set the WORKDIR to the name of the app, since dotnet build currently uses the directory as the dll name  We’re using an environment variable to tell Nuget to skip downloading the XML docs, as we’re not |
| **docker build . -f dockerfile.build -t tradapp:build** | Since we’re using a non-default dockerfile name, we need to pass in the file name. The . represents the context. The directory to pass to the host. It’s the directory used for copy commands.  As we build this, we can see docker create a bunch of image layers |
| **Controllers\HomeController.cs dirty** | Lets make a minor change to one of our controllers. This change needs to be rebuilt |
| **Docker build . -f dockerfile.build -t tradapp:build** | We’ll run the same docker build command, and we get a full build. |
| FROM microsoft/aspnetcore-build:1.0.1  WORKDIR /app  COPY project.json .  ENV NUGET\_XMLDOC\_MODE Skip  RUN dotnet restore  COPY . .  RUN dotnet publish -c release -o publish | But, I said we wanted to benefit from image caching  Lets do a minor change and first restore the project.json file, run the restore, then copy in the rest of the project |
| **Docker build . -f dockerfile.build -t tradapp:build** | We’ll do a quick build, |
| **Controllers\HomeController.cs dirty** | Make a change again |
| **Docker build . -f dockerfile.build -t TradApp:build** | Do another build  Notice the dotnet restore layer was cached:  Step 2 : WORKDIR /Tradapp  ---> Using cache  ---> 42395922525f  Step 3 : COPY project.json .  ---> Using cache  ---> a5657baab911  Step 4 : RUN dotnet restore  ---> Using cache  ---> f84b6f94904b  If we were to change a nuget reference, this layer would be invalidated and the restore would execute |
| ENTRYPOINT tail -f /dev/null | We provide an entrypoint to keep the container running so we can copy the contents out of it |
| **Docker build . -f dockerfile.build -t tradapp:build**  **docker run -d --name tradapp-build tradapp:build** | We’ll build one more time to add our entrypoint and start our container |
| **docker cp tradapp-build:Tradapp/publish/ bin/publish** | Now that we have our build image, that contains our content, we need to get the content out  We’ll use the docker cp command to copy the contents out  We first need to instance the container |
|  |  |

# Summary