Course Overview
Course Overview

Welcome to the Docker for Web Developers course. My name is Dan Wahlin, and I'm a web developer and software architect and really excited about the potential that Docker offers us as web developers. Now, any time you start with a new technology, you want to know the benefits that it's going to bring to you, and we're going to start with that at the very beginning of the course. From there, we'll jump into installing Docker on a Windows machine and on a Mac and learn about the tools and commands you can use to work with Docker, including key Docker client commands you can run, such as docker pull, which will pull images from Docker Hub. We'll learn about what an image is, how you can convert that into a running container, and how the layered file system plays a role behind the scenes. By the time we're done, we'll have an entire development environment set up using something called Docker Compose. And this is a really powerful technology for the development environment, and you'll see the process from start to finish of building a fully functional website. So we have a lot of great stuff to cover in this course. As I said, I'm really excited about the technology, so let's jump into the official agenda for the course.

Why Use Docker as a Developer?

Introduction

Docker gets a lot of attention nowadays and for good reason, but if you've looked into it at all, you might have wondered, what exactly is it? And is it something I can actually use as a web developer? I know when I first started reading about it, hearing about it at conferences and user group talks and things like that, I really wondered if it was something that even played a role in what I did. And the more I dug in, the more I found out that, yeah, actually it can play a big role in our web development operations, and that's what we're going to address in this first module. So we're going to start off by talking about what exactly is Docker, and we'll clarify some key terms and concepts that you need to know in order to be successful to understand how Docker works and how you could use it. Now, from there, we'll jump right into the benefits that Docker could provide us as web developers, and you're going to see there's actually quite a few benefits that it can provide us. There's a lot of great stuff there. Next up, we'll talk about the Docker tools and the role that they each play in this overall development workflow that we're going to be discussing throughout the course. And then we'll wrap up by seeing Docker in action, and I'll actually show an application that's using Docker to hit a database, do some caching, and some other aspects of a normal development workflow and development application. Let's go ahead and get started by answering that all important question of what is Docker, and then jump into the benefits it can offer us as developers.

What Is Docker?

Let's start things off by answering the question, what is Docker? Docker does have some different terms. So we're going to clarify what those are, we're going to clarify where it can run, and how this all kind of works. So Docker itself is just a lightweight, open, secure

platform. This is kind of the official party line, if you will. And the first time I heard that it didn't make maybe a whole lot of sense because I could think of several things that might fit a lightweight, open, secure platform definition. But really what Docker is is a way to simplify the process of building applications, shipping them, and then running them in different environments. Now, when I say environments, of course, I'm talking about development, staging, production, and others that you may have at work. Now, what actually ships with Docker then? Well, we're going to be talking about things called images and containers, and containers are really, really important. You'll see over at the left the Docker logo, and you can think of the whale there as kind of like a ship that has containers. And back in the old days, there was no standardized way to ship things on the old-school ships. So it was a lot more time intensive and not very productive to get stuff on the ship and off the ship. Whereas nowadays, the major shipping companies, of course, have very standardized sized, standard height standard width, shipping containers. So as the crane goes over when the ship docks, it's very quick and efficient, very productive to get those containers on and off these ships. Well, Docker is very similar. If you think of the old days with ships that had no standards for shipping products around, that's kind of where development has been for many years. Everybody does it their own way. Well, Docker provides a consistent way to ship our code around to different environments, and so it's going to provide a lot of benefits that we'll be talking about in this particular section of the module for us as developers. Now it runs natively on Linux or Windows, and when I say Windows, Windows Server 2016 or higher now supports it. We'll talk more about that coming up. And as a developer, if you're on a Windows box or a Mac box or a Linux box, you can use Docker in your development workflow, and it's very easy to get up and running. Now if you're on Mac or Windows, you will need a virtual machine, because by default it's going to expect a Windows server or a Linux server. Now finally, the key buzzwords that are typically thrown around with Docker are images and containers. Let's clarify what exactly is an image and what is a container and how do they relate to each other? So when it comes to the role of images and containers, an image is something that's used to build a container. Now an image will have the necessary files to run something on an operating system like Ubuntu or Windows, and then you'll have your application framework or your database and then the files that support that. So if you're doing Node.js or ASP.NET or PHP or Python, then you'd have that framework built into the image as well as your application code, typically. Now the image itself is not overly useful because it just is the definition. Think of it as the blueprint that's used to actually get a running container going. So if you go back to the shipping analogy, think of some person who does some CAD drawings of what's going to go in the shipping container, maybe even how they're going to arrange it in the shipping container, but the blueprints aren't very useful on their own, but you can use those to create an actual instance of that container. Well, that's the same process in Docker. We'll have images that can be used to create a running instance of a container. Now containers are actually where the live application runs, or the database or caching server or whatever it may be that you need to actually run on a Linux or a Windows Server machine. Now let's dive into the definitions of each of these just a little bit more. So an image is a read-only template. And what it's composed of, and we'll be building these throughout the course, is a layered file system. So you'll have some files, for instance, specific to your Linux operating system or your Windows operating system, and then you'll have your files for your framework, ASP.NET, or Node.js, or whatever it may be, and then they are all put together to make this image. Now once you have an image, you can use that to build this isolated container. And again if you go back to ships, every container is very isolated from the other containers. It makes it so you don't necessarily know what's going on in another

container. Now there are some gotchas there that we can talk about later, but in a nutshell, the image is used to create an instance of the running container. And then you can start the container, you can stop it, you can move it, you can delete it, and it starts and stops really, really fast. And that's what's so cool about this technology is, it's very quick and easy to get a container on the ship and off the ship. And the ship, in our case, would be the development environment, the staging environment, the production environment. Now where does Docker run then? Well, as I've already mentioned, Docker can run on Linux or Windows servers. And so if you're going to be running on a development machine, you have to have a virtual machine, which we'll talk about. Now the exception would be if you're developing directly on a Linux machine, then you could just run Docker containers natively. But Docker ships with what's called a Docker client, and Docker client can then integrate with these different operating systems, such as Linux, and it integrates with a Docker Engine, a daemon that you'll see here. Now Docker itself has its roots in Linux. That's actually where it came out of. The Docker, the company, built on top of some LXC support, it's called Linux Container Support, that's already built in to the Linux operating system. Now likewise, Windows Server 2016 or higher also has Docker support built in, and so the Docker client there could be used to integrate with the Docker Engine, which can start and stop and delete and do all those things with our containers. So think of the Docker client as kind of the commands that are given to the people that load the ship or remove things off the ship, whereas the Docker Engine could be the cranes and the people running those that actually get the container on the ship and up and running. Now what's the difference between Docker containers then in virtual machines? Because this is one of those things that, the first time I learned about this, it didn't make a lot of sense to me. So virtual machines always run on top of a host operating system. So of course you could have a host running Linux or Windows, and then you can run a guest OS on top of something called a hypervisor. And so on the left, you might have, App 1 might be a PHP app, for instance, with its binaries, libraries, whatever it may be, or ASP.NET or Node, or whatever you have. And then App 2 might be running on a different guest OS. So let's say the guest OS on the left is maybe Windows, and the guest OS on the right could be Linux. The bottom line is you have a copy of the operating system for every virtual machine. And so, depending on the type of hard drive and things like that, it can be a little bit expensive to start up and stop a virtual machine. They run well, but they're pretty big. The images for a virtual machine to get it up and running are generally multiple gigabytes in size. Well, let's compare and contrast that with Docker containers. Now they do sit still on top of a host operating system, Linux and now, most recently, Windows server 2016 or higher. And then we have this Docker Engine, which can integrate the containers with the host operating system. And so now, as we get a container up and running, you can think of the host operating system as the ship and then the container for App 1 has everything App 1 needs for that particular feature, so Node.js with all the application code, for instance. Now App 2 might have a completely different container, and then typically applications will have multiple containers. You might have a container for your database, a container for a caching piece, a container for your application code and the framework, those type of things. But the bottom line is they sit right on top of the native operating system. So when I ship these around, I'm shipping a smaller image. It's very small compared to a guest OS virtual machine. They also start containers also start very, very fast, and we'll be seeing that throughout the course. They're great. They just come up in a matter of seconds. The difference here is very, very big. Containers sit on top of the host, whereas guest OSs and VMs sit on top of the actual host, but they're their own copy of all the files. Everything is a copy as you make a new VM. So that's an example of what Docker is. We've now talked about

images and containers, and we'll be delving much, much more into those throughout the course. And then we've also done a comparison between Docker containers and virtual machines. So now that we've done that, let's start talking about, all right, this is great and all, but how does this actually help me as a web developer?

Docker Benefits (for Web Developers)

Docker offers several different benefits to us as web developers, and in this section, we're going to walk through some of the key benefits that we can get by leveraging it. So whether you work on a team of one or many, Docker can help set up a development environment very quickly, and that's really one of the key aspects that we're going to focus on throughout this course. Although that's just a very minor benefit, it's definitely a big benefit as a web developer. Docker can also help eliminate app conflicts. If you have versions of framework saying you can't move to the latest version, isolated containers can help out there. It also provides a way to move your code and the entire environment of the code between your different environments, so between things like development, staging, and production. And if we can do all of that, we can more than likely ship software faster, and that's a good thing, of course. So let's dive into each of these four areas really quickly. So when it comes to accelerating developer onboarding, oftentimes we have multiple team members, of course, and we might have some developers, maybe a mix of designers or people that kind of do both of those things, and oftentimes we want people working with the actual version of the app versus just a prototype that's separate. And so we might have a web server, we might have a database server, caching server, and those types of things, and setting all that up on an individual developer machine, especially for people that work remotely in different scenarios, can be challenging because you have to get the security right, you have to get the configuration settings right, make sure the right versions are on there, and so getting that right and not having surprises after the fact can be a little bit challenging. So Docker can help there because we can make one or more images that can then be converted into running containers, and those containers can run on our different developer and even designer machines. You'll see in just a little bit to get this up and running, you can just run a simple command from the command prompt, so you really don't even have to be a developer per se to get some of the benefits out of what Docker can offer here. Now the next thing we'll talk about that Docker can help us with is eliminating app conflicts and version conflicts. Oftentimes you have an app running on a specific version of a framework, and you'd like to move to the next version of the framework, but you're told you can't because that'll impact other applications running on the production servers. And so what Docker can offer is isolated containers, and each container that actually contains the framework that's having this versioning can actually be isolated, as we've talked about. And as a result, we can have V1, and App1, 2, and 3 that are targeting V1 will run fine in their own containers, and then App1, 2, and 3 targeting V2 can run in their own containers. And now we can have different versions of whether it's Node or PHP or ASP.NET running. This makes a lot easier now to work with versioning and app conflicts. Now some of the frameworks out there obviously have some of this versioning built in, but with this, you really won't have to worry about it. Even if your framework doesn't have a good versioning story as you move between versions, Docker can help out in those scenarios. Now it can also help, as mentioned, with consistency between environments. And this is one of those things that I know I personally have been burnt by over the years. Going way back around the year 2000, we had

a particular project I was working on at a consulting company, and our development environment was set up by the company. We didn't actually do it ourselves, so we had to work on remote machines. And the staging environment was also set up by them, and everything was working great on dev and was supposed to be the same a staging, but turns out it was not, and so we had a nice surprise and had to do some rewrites of things as we moved our first code over to staging. With Docker, we can eliminate a lot of these surprises because we'll simply move the different images that we're going to be building throughout this course over to the different environments and get the containers up and running. And that way, if it runs on dev, it definitely should run the same on staging and production, and we'll talk about how to get all that set up. Now, when doing that, that just means obviously we can leverage all of these benefits to ship code faster. And that's really what software is all about is productivity, high quality, predictability, consistency, you know, all these different words we can throw out. But as we do move our images between dev and staging and production and get those containers going, we can leverage these benefits that Docker offers of the isolation of the containers. We can have a consistent development environment and all the other benefits with versioning and things that we've talked about. So there really is a lot of good things that Docker brings to the table for us as web developers, and now you've seen a few of those.

Docker Tools

Before we can get started using Docker, we need to install some Docker tools. Now the tools that you install are going to depend on your operating system, so I'm going to talk through some different options here. We'll talk through a legacy option and then a more modern option, and that way, regardless of what operating system you're on, you should build to get Docker going. The first one that we're going to talk about is called Docker Toolbox. This is a legacy option that at this point is really only for Windows 7 or 8, or if you don't have Windows 10 Pro, you just have Windows 10 Home, then you possibly might use it there as well. Now this provides all the image and container tools though that you would need. So while it's an older legacy option at this point and it's been deprecated, it does provide the key tools that you need. So if you are are on one of these operating systems, then you could at least get it working. The way it works is it uses a virtual machine called VirtualBox to run a Linux VM, and then the Docker commands that we're going to learn about are going to execute against this Linux VM that's going to be running in VirtualBox. So what do you get with Docker Toolbox? Well, the first thing you're going to get is the Docker client. This is going to be really important so that you can communicate with containers and images and all these different technologies in Docker that we're going to be talking about. You're also going to get a tool called Docker Machine, and this will let you hook up to the virtual machine. Docker Compose we'll have a whole module on later in the course, but this provides an orchestration mechanism for bringing up many containers, 1, 2, 10, or 50 if you wanted, on your machine. And then another tool is called Docker Kitematic. This is more of a GUI tool, and it's not one we're going to spend a lot of time on in this course, but I will do a quick demo and give you the basics of how it works. And then finally, I mentioned that this is all running in a virtual machine called VirtualBox. And this is something that actually can stand on its own. You don't have to have Docker for VirtualBox, but that's what they use kind of out of the box to run this. Now what if you're not on Windows 7 or 8 though? What would you run? Well, in that case, you're going to run Docker Desktop. So this is going to be for Windows 10 Pro or higher

or Mac. So if you're on one of those, you'll definitely want to pick Docker Desktop. It's much more modern, it's updated frequently, and it's going to have all these tools at your disposal. So this also provides image and container tools, but behind the scenes, it actually uses Hyper-V on Windows. That's why you need Windows 10 Pro in this case. Or if you're on Mac, it uses HyperKit to run these Linux VMs. So that functionality right there will be one of the deciding factors on which one you'll be able to choose, Docker Toolbox or Docker Desktop. If you're on Windows and don't have access to Hyper-V, then you're going to have to choose Docker Toolbox. Whereas if you're on Windows 10 Pro or Mac, then you're going to want to go with Docker Desktop. So as mentioned, this works on Windows or Mac. Now on Linux, you can run something called Docker Engine, and it really depends on the flavor of Linux. There's multiple options if you go to the docs that you can learn about. But you can also run the Docker Engine there, but it's not the full Docker Desktop like you'll have on Windows or on Mac. So what do you get with Docker Desktop? Well, you're going to get a Docker client again, Docker Compose, which is this orchestration mechanism for bringing up and managing multiple containers, and then you also have access to this GUI tool called Docker Kitematic. Now it's not installed by default, but it will run as long as you have Docker Desktop installed. So to wrap up our discussion of the different tools, let's do a quick comparison side by side. So if you're on Windows 7 or 8, you're going to have to choose Docker Toolbox. That will install VirtualBox, and it's going to use a tool called Docker Machine to allow you to connect from your command line into the running virtual machine. If you're on Windows 10 Pro or higher or you're on Mac, then you're going to be using Docker Desktop, and this is the modern version of Docker. And this uses Hyper-V if you're on Windows, that's why you need the Pro version of Windows 10 or higher, and it uses HyperKit on Mac. Now on Linux, although you don't install Docker Desktop, you can install Docker Engine and Docker Compose separately. So it's possible to get these same types of tools but you're not going to have some of the other functionality that I'll be showing you here that you would have on Windows or Mac. So that's a quick summary of the different tools that are available. Let's now switch our focus to Docker in action.

Docker in Action

Now that you've seen what Docker is, some of the benefits it can offer us as web developers, and a few of the Docker tools that are going to be involved throughout this course, let's take a quick look at Docker in Action. So the demonstration I'm going to show you actually uses six containers. We have Nginx. This is a reverse proxy. We have three Node.js instances. MongoDB is the database, and Redis as a caching server, and I'll be able to get this up and running quite quickly on my machine. So let's jump in and I'll show you how this works. Let's assume I've been tasked with getting my development environment up and running, and I needed to look not only like the other team members, but also, like our staging and production environments. Now, if you've done this very much, you'll know that that can actually be a little bit tricky, but with Docker it's very, very simple. So I've already configured some Docker images we're going to be talking about throughout the course, and I already have some containers ready to go. So I'm just going to run a simple command, that we'll learn about later, called docker-compose up, and this is a way that I can basically start up six containers that I need to run this particular application. So we'll go ahead and let this run and it'll take just a moment to fire up, here, as the web servers connect to the database and things. And right now I have an IP address that I already know Docker is going to give me, and

I'm going to hit refresh, and you'll see right now it's not quite ready, so let's go on back, and we should be really close here now. All right, looks like we should be good to go now, so let's hit it now and there we go. So this just hit a website that's using, again, Mongo, Redis, Node, Nginx, some other features behind the scenes, and this is actually a company site. You're going to get to work with a subset of this site, so you can have a more realistic demonstration to work with throughout the course. But it's a pretty standard application. I can go in and get information about different things, go back to the home page, pretty standard stuff, and Docker made this really, really easy to work with. Now I can do a Ctrl+C here, and now this is going to stop all the containers, and from there, we'll be kind of done and ready to go. Every now and then we'll throw an abort message. You kind of just ignore that because we can just start it back up. It looks like we're good. And I can just wipe that out, and if I want to start it back up, we're ready to go there and we'd be up and running. So that's an example of Docker in Action on a Mac. We can get the exact same environment going on the Windows side as well, so I'll go ahead and run a command. We'll start everything up. This will start up in genetics and some web servers and database and more. And once this is all done, because we have the exact same environment that I showed on the Mac side, we'll of course get the exact same output in the same website running here. So there's not going to be any variability between the two sides of the house, and the same goes as we move things between environments such as development, staging, and production. So this is almost done firing up for the first time, and we'll go ahead and try to load this at this point. You can see we can get the exact same website loaded up, and that allows us to very consistently run our code in different environments. So that's an example of running Docker on the Windows side.

Summary

In this module, you've learned what Docker is and seen how it can simplify building, shipping, and running applications across different environments. We talked about that it runs natively on Linux and now on Windows Server, but that it's not the same thing as a virtual machine. In fact, it's very, very different. It can be a lot faster in many cases as well. Now for us as web developers, there's a lot of key benefits that we also discussed. One of the big things is you can get your environment up and running very, very quickly and in a consistent way. This allows developers, whether they're on the team already or contractors or new hires, to get up to speed very quickly, whether it's on Mac, Windows, or Linux, and not have to do some custom installs of various software. We also talked about if you work with multiple apps and multiple versions of apps and frameworks, Docker could help because of the container isolation. We talked about moving code between environments, development, staging and production, for example, and how Docker can provide a consistent way to do that. And all of this really leads to us shipping faster. We can ship our code in a production faster, hopefully make everybody more happy along the way. So I'm really excited about Docker because it offers some things that we really just haven't had before. As I mentioned, it's kind of like shipping in the old days across the ocean without containers to moving into consistent standardized containers for packing and shipping everything around. Well, when it comes to software, we now have a way we can containerize, if you will, our code, and our frameworks, databases, and other things, and this provides a much easier way to ship these different things around between our environments. So as we move along in the course, we're going to jump right into getting Docker installed. You're going to see how to use the different

commands and how you can actually work with Docker in your web development environment.

Setting up Your Docker Environment

Introduction

In this module, we're going to take a look at how we can get our Docker environment set up so that we can work with the different images and containers that we're going to be discussing throughout the rest of this course. So we'll start things off by talking about how to get Docker installed on Mac, and I'll introduce something called docker-ce, Docker Community Edition. Now, we're also going to look at how you install Docker on Windows, and with Windows, you actually have to choose a different version of Docker. So if you're on Windows 7 or 8, you're going to be installing Docker Toolbox. Whereas, if you're on Windows 10 pro or higher, you can use Docker Toolbox, but most people are going to want to go with something called Docker Community Edition, a very similar version to what people on a Mac would want to run. Now, once we explain those differences, talk about how to get things installed on Mac and Windows, we'll also talk about something called Docker Kitematic. Now, most of what you do with Docker is command line, but Docker Kitematic is a GUI type of tool that will let you view images that are up on something called Docker Hub, pull those down to your machine, and then run them as containers. So we're going to introduce what Docker Kitematic is, and then I'll show you a quick example of Docker Kitematic in action and how we can actually pull down some different images out there and get them running on our machines. So let's go ahead and get started by discussing how to get Docker installed on a Mac.

Installing Docker on Mac

Let's take a look at how we can get started installing Docker on a Mac. So the first thing I'm going to do is come over to docker.com, and then we can come to Developers, Getting started. That gets me to this page. That menu certainly may change, but otherwise, just do a little search for Docker Desktop. Now scrolling on down, you'll notice that I have Docker Desktop, it mentions something called Docker Hub, which we'll talk about later, and then you also have an online tool you can get too called Play with Docker. It's kind of a browser-based way to interact with Docker and a command line right through your browser. Now, with Docker Desktop, though, you'll notice that if I hover over it, I can download for Mac, Windows, or if you're on Linux, you can download Docker Engine, and by clicking there it would take you to the different flavors of Linux, and you can pick the appropriate engine. Now, for this particular example though, I'll of course choose Download for Mac. And now that this is done, we can simply double-click it, this will extract it and get it ready to run through, and then setup's pretty straight forward you'll see. Alright, so I can go ahead and drag that over to my Applications. That will take a moment as it copies it. And then from here, I can go ahead and go into my apps, and we can actually click on it to fire it up. So I'm going to come on in, and I'll just search for Docker. There it is. Click it. All right, now this will say, hey, I've downloaded it off the internet. Pretty standard. We'll hit Open, and now you'll notice up in the corner here it's firing up. Now, I've already had Docker on this machine, so this will be slightly different

than what you might see, but once this whale is kind of done spinning up here, you should see that Docker Engine has started. There we go. And now if I click on it, you'll notice that Docker Desktop is running and that I have some different menu items here. Now, one of the more important menu items you're going to see is Preferences on Mac. So let me go to that real quick. Now you'll see right off the bat that we can start Docker when we log in, we can automatically check for updates, include backups, send usage statistics, all that fun stuff that you kind of see with a lot of applications. Down here in the corner you'll notice that Docker is running. Now, I also, because I've already had this running on this machine, have something called Kubernetes that's running. This is a way to run containers and orchestrate them in more robust production-type scenarios. Check out my Kubernetes for Developers course if you're interested in more on that. Now coming back up, we can go to Resources, and from here I can say, hey, how many CPUs does Docker get? How much memory? What about the swap disk? And I can adjust these things so that I don't eat up too much hard drive space and too much memory because you can really max it out, and it will really push your system to the max. Or you can really minimize it, and maybe Docker won't be quite as fast or high performance, but it would at least work if you have less memory on your machine. Now we can also come on in, and you'll notice I have FILE SHARING. This is going to be for something called volumes. We're going to have a whole module on that coming up a little bit later in the course, but this is basically a way for a running Docker container to actually point back to a folder that's going to be on your local machine. If we have to deal with proxies, we could set up proxies here and even some network configuration, which you normally don't need to mess around with at all. Now coming on down. we have Docker Engine. This is one you normally don't mess with at all. We have Command Line features for experimental up-and-coming features. We're not going to turn that on. And then if you wanted, you could turn on this other option called Kubernetes by checking this Enable Kubernetes check box. We're not going to worry about that one here, but feel free to play with that if you'd like. So that's what you're going to get out of the box, and what we can also do is every now and then, you might want to visually see what's going on with your containers. So notice as I exited that screen, I get to another dashboard screen, and currently I don't have any containers running, but it says I could run this command right here. In fact, we can just copy it, and then we could go to a command line, so let me go ahead and do that. Let me go ahead and paste that in now, and we'll go ahead and run it. Now this just pulled an image that wasn't available on my machine from something called Docker Hub down to my machine, and it made this image available. And we'll talk more about what that is and how you build custom images as well throughout this course. Now from here, let me close this, we'll come on back to our dashboard app now, and notice that we have this condescending_ aryabhata, I'm not sure how you say that, running on port 80 here. And we can click on it, we can get log information, right now we don't have much, we can inspect the container, get information about it, get statistics about it, and this is all being done visually through the dashboard. Now I can also come up and stop the container, restart it, go ahead and stop it again, and then we can delete it if we wanted, and I wouldn't even have to worry about any of these commands that we're going to learn later. Now, I'm going to offer you want to know about these commands, because we're going to be using them a lot, but this is a really nice feature of Docker Desktop. It gives you this visual dashboard. So that's an example of how to get it up and running on Mac. Now let's switch our focus to Windows.

Installing Docker on Windows

Now let's switch over to the Windows side. Recall earlier that I mentioned that there's a few options for Windows depending on what version of the operating system you have. If you're on Windows 7 or 8 or even Windows 10 Home, then you're going to have to go with Docker Toolbox because you won't have access to Hyper-V. If you're on Windows 10 Pro or higher, then you will have Hyper-V accessibility, and therefore you could install the preferred version, which would be Docker Desktop. Now I'm going to walk you through how to get started with both, but we're going to focus on Docker Desktop in the install and throughout this course overall. So the first thing to talk about is Docker Toolbox. And as you can see on the web page here, this is the legacy desktop solution if you don't have Hyper-V available. This will run VirtualBox instead. And the good news is you can still run images and containers and do all these commands we're going to learn. It's just a older way to do it, but it definitely works. Now you can go to docker.com/docker-toolbox. And from here down at the bottom, you can install it for Windows, and then they have some instructions. Now I'm not going to be running that one here, but I'll let you if you need that. Go to this link, get it installed. If you need any help with that, refer to the install instructions that they provide. Now for Docker Desktop, it's going to look identical to what you saw on Mac. We can scroll on down. We can click Download for Windows. Once this executable comes down, we can simply run through the install routine. All right, so now that's done. Let's go ahead and open the file. I'll go ahead and approve doing that. And now this is going to start a download routine that will get everything we need. Now in my case, I already have this installed on this machine. So it's going to be really fast, as you see here. It says existing installation is up to date, and I'll just close that. But for you, you're going to have to run through it. It'll take a little bit of time to get it going, of course. Now from there, we can then go run Docker. So if we come on down to the Start menu, we can type docker, and there's Docker Desktop. Now what'll happen is this will go down in our tray in kind of the right-hand corner here. So let me go ahead and go to that. And you can see that our whale is currently starting up, and we now have a message that says Docker is starting. Let me click that again, and then we'll just wait a little bit for this to get going. All right, so now that's done. We can do much like we did on Mac I can right-click. Instead of Preferences, I can come up to Settings. And the settings are going to look pretty identical to what you see on the Mac side. There's a few differences here and there, but it's very, very similar now. So we can start once we log in, automatically check for updates. We can even expose some TCP ports here and do even more if you have Windows subsystem for Linux going. We can come down to resources, set our CPUs, our memory, our swap, and image size. Turn on volume sharing. Notice that I have C drive exposed here. And again, we'll talk more about what a volume is and why you'd use it with containers a little bit later in the course. If you deal with proxies, you can configure that. And then just some general networking here. Now Docker Engine, this again is something we normally don't have to touch. Same with command line, more experimental in this case. And then I've already talked about there's this other tool called Kubernetes, which you can also enable here if you'd like to. Now if we cancel out of this, we'll get to the dashboard again. And this time, notice I have quite a bit going. They're all stopped, but I have quite a lot going here you'll see. So I could actually start to just clean these up if I wanted. We could start deleting these, but I'll go ahead and leave these. But it's very much like what you saw with Mac. I can actually come on in and start my containers, stop them, delete them, and do more, get to the logs and other things we saw. So as an example, we could click on maybe this ASP.NET PostgreSQL Docker one, and notice it's exited. But I could start that back up. Then we could get to the logs and

do all kinds of fun stuff there. So there's a lot of great stuff you can do with this dashboard. And although we're going to focus on the core Docker commands, knowing about this is nice. I don't use it every day, but every now and then it's just easier to come into here to do certain things. So that's how easy it is to get either Docker Toolbox or Docker Desktop going on Windows.

Getting Started with Docker Kitematic

Now that we have Docker installed, we can start working with the different command line tools that it provides. Before we do that, though, I want to talk about an additional tool that's available called Docker Kitematic. And although we're not going to be using it throughout the course, it's a great way to visually see what are images and what are containers, and you can even pull down images and start up containers right there on your machine very, very easily. Let's take a quick look at what it offers. So Docker Kitematic is a GUI tool that makes it really easy to work with images and containers. And as I mentioned, we're going to be using command line for everything throughout the rest of the course, but I like Kitematic simply because if you're new to Docker, really haven't played with images or containers, or at this point don't even really know what those are, so we'll be providing more details as we move along, then this will provide a really easy way to get started. Now, it allows you to visually search on something called Docker Hub for these images, and the image can then be downloaded to your machine, and then you can create, run, and manage containers using this GUI. So just with a few clicks of the button, you can actually issue commands behind the scenes that we'll be learning about later that you would normally have to do on the command line, but Kitematic kind of hides that from us. So if somebody wanted to play around with images and containers and doesn't know the cmdlet at all or really doesn't use it, Kitematic would certainly be an option, but I think even if you are going to be using command line, then it's nice to know about at least and interesting to explore. So let's go ahead and take a look at Docker Kitematic in action.

Docker Kitematic in Action

So let's see what we can do with Docker Kitematic and take a look at how we can install it and then from there use it to search images, pull those images down to our machine, and even run containers. So the first thing we'll want to do is go to this GitHub URL, github.com/docker/kitematic/releases It's important to note that you can get Kitematic through Docker Toolbox, but if you want the latest version of Kitematic, you would want to go to here. Now once I've done that, I can download for Mac, Ubuntu. I can download for Windows as you can see. We'll do Windows in this case. And once this is done, we can open it, and you'll notice I have a bunch of files in here. So from here, it's really just a matter of extract these to a folder of your choosing. So I'm just going to select all these. We'll go ahead and copy these. And I'm going to go to my desktop for this case and just put a folder here. We'll call it Kitematic. But you, of course, can put this wherever you like, and then we'll paste these in. All right, now from here, we just need to find the exe, and you'll see that right here, Kitematic.exe. So we'll go ahead and click this. You may be prompted with a message. Just click on More info if you're on Windows. I'm going to hit Run anyway. It just doesn't know what this program is. And here we go. So from here, I could log in to Docker Hub

if I had a Docker Hub account, which I do, but you may not. So we're not going to worry about that. You could sign up though if you'd like and get an account, and this would allow you to store your images, your Docker images, up in Docker Hub. Now I'm going to hit Skip For Now. Once Kitematic is up, now it's on us to find the image that we'd actually like to run. So we could choose a Hello World Nginx, Ghost for blogging, Jenkins, Redis. You can even do a Minecraft server if you'd like if you have the client and on and on and on. So you'll see there's a lot of different options. Now I'm going to go ahead and just type Nginx here. And there's the Hello World. But I'm going to go get the official Nginx right here. And notice if I hit ..., I can get a little bit of info. So I can get what the selected tag I want to do is. We could filter. I'm going to go ahead and just leave the latest here, but notice I can pick different versions of it. That's kind of nice. And then to actually download the image and run it, we can just hit Create here. Now this is going to go off to Docker Hub and download the Nginx image. This will be the latest version. Now that that's downloaded, it's going to go ahead and actually start up the container now. And then once this is all ready to go, then we can actually use this to call it. Now notice up top, it says it's running. I could stop it, restart it, exec into it, we'll learn about what that is later in the course, even view documentation about. If we click on this, it will take us up to Docker Hub. And you can see that we can get information about Nginx. Here's some of the different versions we have. If we scroll on down, we'll even see how to use it from the command line. Now what's nice about this though is that with Kitematic we can do all this without actually typing the command. So let's go back to Kitematic now. And if I come on over to Settings, notice I can get some general information. So we can get some environment variables it's using in the container, the host name and ports. So it looks like there's a port 80 running in the container, but we can call 32769 to actually call it from outside through the browser, for example. We'll learn later about something called volumes, this one doesn't have any. Networking, we'll talk about what bridge networks are, and even some advanced features. Let's come on back to the ports though, and I'm going to copy this. And then let's run to the browser here, and we'll go to this localhost 32769. And there we go. And now we have Nginx up and running. What's great about this is number 1, we really didn't have to know anything about Docker. We could just use Kitematic to help get us started. But number 2, it makes it really easy to start these containers, restart them, stop them, delete them, and I'll show you some of that right now. So let's go back to Kitematic, and let's just go back to home here. So first off, we can see that the browser hit it. Notice we have some logs here. And then from here, I could stop it. We could start it back up. Notice it's running again. We could restart it. We can shell into it or exec into it. That opens a command prompt here, and now I could do things like show me all the files and folders. And then you already saw we can view the docs. Let me go ahead and stop it though. Here it is. And then notice, we have a little X here. If I hit that, we can now remove it. And there we go, it's gone. So that's an example of some of the fundamentals of getting started with Kitematic. Now I mentioned earlier, we're not going to be using this throughout the rest of the course. We're going to learn how to do the actual commands. But it's a great way to get started and play with Docker images and containers and see how you can get started using them.

Summary

In this module, we learned about different options for getting Docker running on different operating systems. We started off by talking about a legacy version that's available called

Docker Toolbox, and that would be one you might choose if you're on Windows 7 or 8 or even on Windows 10 Home Edition because that won't have support for Hyper-V. Therefore, you're going to have to go with VirtualBox. Now if you do have Windows 10 Pro or higher or you're on a Mac, then it's highly recommended, you run Docker Desktop. If you're on Linux, you can install the Docker engine that I mentioned to paint on the version of Linux that you're running. Docker Desktop, as we saw though, gives you not only direct integration with Docker, but also a dashboard. You can customize different preferences and settings. So you can set, for instance, the CPUs that are going to be used, memory, swap, size of the virtual machine, all that type of information. Not only is it very easy to get going, but it also makes it very easy to interact with. And so we'll be focusing on Docker Desktop a lot as we move throughout the rest of the course. We also took a look at Docker Kitematic, and although I personally don't use it on real-life projects, it does provide a really, really nice starting point if you're brand new to Docker. Maybe you've seen nothing about it. This is the first time you've learned about images and containers, and you just want to get something going quickly and easily, then you could use Docker Kitematic to do that. And as you saw, it provides a nice visual way to integrate with downloading the image from Docker Hub and actually running the container, seeing what ports are being used on that container, and more. So now that we've talked about how to get Docker and the various tools going on our systems, let's go ahead and move on and dive a little bit deeper into more Docker features.

Using Docker Tools

Introduction

In the previous module, we talked about some of the different Docker tools that are available and what operating systems support them. We're going to review that upfront here, but we're going to dive right into some of the different commands that you can use with these tools to interact with the virtual machine or to interact with images or containers. Now as a quick review, we talked about how we have Docker Toolbox or Docker Desktop. Docker Toolbox, again, is the legacy version, the older version, Docker Toolbox is the newer and preferred version. Now both of these provide what's called a Docker Client to let us interact with Docker images and containers, and we're going to learn about some of those commands so that we can use the Docker Client tool. Later in the course, we're going to talk about another tool that's built in, called Docker Compose. This is used to bring up one or more containers and do what we call orchestration, and you'll see that we can do other things with it as well, including building one or more images. Earlier, we looked at Docker Kitematic. This is another tool that's available, and this provides that GUI that lets us pull down an image and actually run a container without having to know these different commands that we're going to learn about in this module. And then finally, if you are running it on Docker Toolbox, the legacy version, you'll have to use something called Docker Machine and VirtualBox. They will be installed automatically if you have to go to that. If you're on Docker Desktop, you're not going to need those, though. So for those that might be running Docker Toolbox, the legacy version, I'm going to start off with an introduction to Docker Machine. Now, if you're on Docker Desktop, you'll be able to skip these videos because you won't actually run these commands. You won't use it because it's not something that's part of Docker Desktop. But if you are on Docker Toolbox, I'll show you how to get started with it, and then I'll show you a few examples

of Docker Machine in action. Now, whether you're on Docker Toolbox or Docker Desktop, you'll definitely want the next part. Now we're going to switch our focus to Docker Client, and we're going to start learning about different commands you can use to work with images and to work with containers I'll provide examples of that again, show these in action, and then we'll do a review at the end to talk about some of these different commands. So let's go ahead and get started by talking about Docker Machine, and once again, this is only going to be if you're using Docker Toolbox. So if you're using Docker Desktop, feel free to skip the Docker Machine videos.

Getting Started with Docker Machine

Let's take a look at another tool in the Docker toolbox called Docker Machine. Now Docker Machine can be used to create and manage your local machines that you're going to be working, for instance, on your development environment machine. It can also be used to create and manage different cloud machines, such as ones on AWS or Azure or other cloud-based providers. But we're going to mainly be using it to manage our local machines. Now, as mentioned, if you're on Mac or Windows, you are going to have VirtualBox because Docker out of the box is either going to be running when the Docker containers run, I should say, on either Linux or on a Windows server. Now we're going to mainly leverage the Linux features here. And so for us to interact with that, we need a way to host it. And that, of course, is what VirtualBox does. And so Docker Machine will let us start and stop and create different virtual machine images. Now it can also configure the environment so that when you pull up a command line bash type shell in Windows or on Mac, that you can use the Docker commands to manage your images, start and stop your containers, and perform those types of operations. Now there are a few commands that we need to know to get started. And I'm just going to show you a quick list of a few of the key commands. These are not things you need to memorize because I'm going to be using these throughout the course. But they are good to know. So one of the commands is called docker-machine ls. Now Docker Machine is the actual command line tool, and then ls is the command we're going to run. So this would list all the different machines that we can issue Docker commands against. Now what do I mean by machine? Well, out of the box, you're going to see in a moment that when you install Docker, you're going to get one VirtualBox machine set up called default. Now you can certainly set up others, but when you first get started, one is good enough. And so we'll have default, and we can use Docker Machine to list that and any others that you might create. That's what the ls command does. Now we can also start and stop our virtual machines, and so dockermachine start, and then machine name would be whatever it is. As mentioned, default is the default name of the machine, so we can use the start command or we could use the stop command. That's how you can easily start and stop one of the VirtualBox images on Mac or Windows if you'd like. We can also configure the environment for a machine. This is really important, and I'll be showing this in just a moment. But when you first pull up a command line terminal window, you're going to want to issue some Docker commands to manage your images, your containers, and things like that, and you first need to make sure that Docker knows what machine it's going to be interacting with during that terminal session. So you'll see in a moment I'm going to use docker-machine env command to do that. Now we can also get the IP address of a given machine, and that's useful as we start to test our containers that are running. For instance, we might pull up a browser and want to call into the machine and call a specific container, and we'll be demonstrating that as we

move along as well throughout the course. So these are some of the key commands that you can use for Docker Machine. There are certainly others, but these are the ones you need to know to get started. So let's take a look at an example of some of these commands in action.

Docker Machine in Action (Mac)

Once we have Docker Toolbox installed, we can get directly to the Kitematic tool that showed earlier, but we can also get to the Docker quickstart terminal. And this is a terminal window, a command line window, that you can get to to interact with Docker tools such as docker-machine. Now what I normally do is drag and drop this down to my doc, and you'll see already have it here. So I'm going to close this one. We'll come on down and open it up. And the first time this fires up, if your virtual machine is not running, your VirtualBox machine that Docker Toolbox and stalls, then it's going to go ahead and run a machine, and it's going to give it a name of default. You'll see that right here. And it's going to take a moment to start this machine up. Now from here, it's going to copy over some certificates and do some other configuration. And once you get to the nice little whale image, you'll know that Docker is now configured as it says to use this machine called default, and it even gives you the IP address of the machine that VirtualBox is actually hosting. So what is default? Well, default is our VirtualBox Linux machine that we're actually going to be issuing Docker commands against. But before we do that, the first thing you want to do is make sure that your terminal window is linked up to the proper machine. Now ours obviously is because I have my default machine is all wired up here it says. And once that's done, we can start working with these machines and using them. So one of the key things you'll want to know about is things like the IP address. And you'll see that, yeah, you can see it here. But as you start issuing commands or maybe you type clear and clear it all out, you might forget what that is. So there's a couple ways we can get the IP address of the running machine that is hosting our Linux server in this case. And one of those is we can say docker-machine ls. And the ls command will automatically let us know all the machines that are running on this particular box on my development machine. Looks like I have one called default. It's running through VirtualBox. It's up and running, and there's the IP you can see. If I just wanted to get the IP address though, I could say docker-machine ip for the name of the machine. Now you would have to know the name by default. It's called default. But you can create other machines as well. Now we're not going to do that. We'll just use the default machine in this course, but it is possible to create others. So I'm going to hit Enter, and you'll notice I can get the IP address. Now, likewise, if I just want to get the status, you'll see the state which is running. And if I want to get it for a particular machine, we could say docker-machine, give me the status for default, and it's up and running. Now from here, I can also start and stop machines. Now this one is obviously already started, but we could say docker-machine stop default. And this will take a moment to run, but this will actually shut down the running virtual machine. And typically, if I'm not using Docker on a particular day, I will shut it down. You'll see it was pretty quick to do this because that'll free up some memory on your machine if you happen to need it. But a lot of the time I'll just leave it up because I'm jumping in and out of Docker throughout the day when I'm on a particular development project. Now we can also say docker-machine start default, and this will now start the machine back up. Now, once again, when you run the quickstart terminal, if it's not already started up, it'll start it up for us. So you kind of don't have to use the start as much. But every now and then you might shut it down yourself and then want to manually restart it. Now while this is running, I'm going

to go ahead and open up just a regular command terminal window here. All right, so I already have my running Docker Machine one, but I'm going to do just a new window. Now because I didn't use the quickstart terminal, it didn't run any of the early shell scripts that kick us into the world of Docker. I'm just in normal terminal mode. In fact, let me just make this a little bigger so we don't get confused by the one in the background. And now let's go ahead and try to do something like docker-machine ip of default. And you'll notice I can get to that, but if I start to run commands, and we'll learn about some of these Docker client commands a little later in this module, but I'm going to do one called docker ps, and you'll notice I get an error. Cannot connect to the Docker daemon. Is the docker daemon running on this host? And might, the first time you see this, do what I did and go well, wait a sec, I know the virtual machine is running, so what's the problem here? Well, if you go through the quickstart terminal, you probably won't have to do this. But if you want to either A, configure a different terminal to use the default machine so that you can issue Docker commands against it such as this ps, ps would list all of our containers, by the way, but we'll learn about that coming up, or B, we might want to hook this terminal up to a different machine other than default. What we can do is wire up this terminal to the machine that we want to issue Docker commands against. And the way we can do that is through another command called docker-machine env. And then if I just do this, we'll get an error. But we have to tell it the machine name. So we'll say default. And what this will do is add some--- You'll see kind of variables here, into our environment variables. And then it's going to say run this command to configure your shell. Now when we ran the quickstart terminal, it's already doing this behind the scenes to hook us up to the default because that's the one you get out of the box. But if I wanted to either hook up to a different machine, then I could have said docker-machine environment whatever that other machine is, my machine maybe. And then what you have to do is run this eval command. So you literally just copy this, paste it down, and then run that. And now when I run docker show me all the containers, which is the ps command, you'll notice that at least it works. Now I don't have any. We'll be doing that shortly. So that's a really, really nice tip that I know I struggled with initially when I got into Docker because I didn't realize that you had to hook up the terminal window if you didn't use the Docker quickstart terminal anyway to the actual machine that you want to issue the commands against. So that's a quick look at the docker env command as well. All right, so now we've seen several of the commands. You can actually list all of them by saying docker-machine and just hit Enter, and this will list all of the different commands that we have available. You'll see there's quite a list here. A lot of stuff you can do, but we're now kind of going over the key ones, the environment command, the ip, the status, the ls, the start and the stop. You can even restart a machine. Very similar, just docker-machine restart default. And there's even ways you can create new machines. If you wanted to have different versions of Linux or something like that running, then potentially you could create a different machine if you'd like. So that's a look at some of the key Docker commands that you can run that are specific to Docker Machine. And again, Docker Machine is part of the Docker Toolbox that we've already installed, and now we have that up and running. And we can now interact with that machine, and that's what it looks like from a Mac standpoint.

Docker Machine in Action (Windows)

Whether you're working on a Mac or on Windows, you can run the same exact Docker machine commands. In fact, you'll run a Bash shell even if you're on Windows as you'll see in

a moment. Now we've already installed the Docker toolbox so it adds some icons on my desktop area. What I like to do is drag down the Docker Quickstart terminal down to my toolbar, so I'm going to go ahead and open that up, and you can see that it automatically fires up the virtual machine called default. This is running as a Linux virtual machine in VirtualBox and gives me the IP address. Now, one of the things that's a little bit different, though, on the Windows side is that you'll note that I'm not in a normal DOS mode here. I'm in a type of Bash shell, and so I can run commands, for instance, like ls, which would be very similar to dir that you're used to in Windows and that'll list where I'm at all, the files and folders and things. I can type clear kind of like cls in windows and notice that clears off the screen and there is a lot of other things we could do that are related to more of a Bash environment. So what this does is it installs this Bash environment for Windows and that's a good thing actually because these same commands that we can run on Mac and Linux, you can run here on Windows. So let's get started by jumping into the Docker machine command itself. Earlier, when I pulled this up, you saw that we had a Docker machine called default, and again, it listed the IP. And so, I can list all the machines on my Windows environment by saying docker-machine ls and this will now list that I have a machine called default, it's running on virtualbox, the status is it's up and running, and there is the IP address you can see for that machine. Now, if I wanted to get to the IP on windows for that particular machine, we could say docker-machine ip for the name of the machine. So you simply take that name there, hit Enter, and there we go, and I could also get the status by doing docker-machine status for default, and you can see it's up and running, so ls will get you all the machines, but if you just want to get a particular property of that machines such as the IP or the status, then you can run those commands as well. Now, when we ran the quickstart terminal, this actually ran some behind the scene scripts that made it possible to run other Docker commands against this machine called default and one of the commands we're going to learn about later is docker ps. Now, if you watched all of this for the Mac side of it, this is going to look exactly the same and that's kind of the point of it, but if you are just jumping right to this Windows information, Docker PS will list the running containers that we have and we'll talk more about this coming up later in the module, but you'll notice that it works, alright, and the reason this works is because when I did the quickstart terminal, it already made sure that my default virtual machine was running, if it wasn't, it starts it up, and it hooks this terminal window here to that running machine so that when I issue commands, many of which we'll learn about a little bit later here in this module, such as docker ps, those commands work. If we weren't linked to the machine, then we'd have some problems and we couldn't run the commands. Now, one way you can switch machines or link it up yourself is you can run docker-machine env for the name of the machine that you want to hook to and what this will do is it'll set up some environment variables for us, you'll see those there, and then it tells me go ahead and run this eval statement. Now you'll notice there is a space here, and so, unfortunately, if you run it as they sort of show you, let me just copy that down, we'll paste that, we're going to get an error because of the space. It didn't know what to do. Now we could just kind of abbreviate this and run docker-machine directly because it is a known command, you've already seen that up here, but if we want to leave what they give us and just fix it up, I'm going to hit home to go to the beginning and I'm just going to add a single apostrophe and we're going to wrap this path, let me do and and we'll kind of back it in here. Alright, now it'll know that we have some spaces in our path, in this case, in our folders, and now it's going to work appropriately. Now we were already hooked into default so that was a very redundant thing to do, but it's good to know because if you ever for whatever reason want to switch to a different machine, you'll have to set the Docker environment to something other than default, maybe

my machine if that's what it was called. Alright, so the last thing is if you want to see all the commands Docker machine has, you can say Docker machine and this'll list everything you can do do. So you'll see there are quite a few commands here. We can go in, and in fact, I'm going to show a start and a stop to wrap up, but we can start a machine, we can stop a machine. I've shown you the status, the ls, and the ip, but there is a lot of other things you can do. You can actually create new machines. We're not going to do that because the default machine has everything we need for now, but there is a lot of information. You can inspect and kill a machine and do all kinds of fun stuff. We're focusing right now on the key commands that you need to know to get started here. The last thing I want to show you is that you can also come in and say docker-machine start or stop. Obviously, default is already started. We've seen the ls and we can see it's running. But I could also come in and say docker-machine stop default. And now, that's going to stop that virtual machine, and then likewise, I could say docker-machine start default and there is even a restart. So let's go in and now that that one is stopped, which was pretty quick, you could see, we can start default, and this will take just a little bit more time to start it back up, but then we'll have a running virtual machine again and we'll be kind of ready to go. Now if when you start a machine, if for some reason when you issue Docker commands, you get a error, then you'll need to run that Docker environment, the env command that I showed and then do that eval copy and paste thing. So that's an example of how you can run Docker machine commands on a Windows environment, and again, if you happen to watch the earlier one on how to do this on a Mac, you should now see that the commands are the same, which is really, really nice. It doesn't matter what your on Linux, Windows, Mac, you're going to be running the same exact commands. And so once you learn these core commands that I'm going to focus on here, you're pretty much good to go regardless of what operating system you're going to be running against.

Getting Started with Docker Client

Once we have Docker Toolbox or Docker Desktop installed and running, we can start to use something called the Docker Client. Now Docker Client is a way to interact with the actual engine behind the scenes or Docker daemon, you may hear, and this will allow us to run commands that can then interact with images, Docker images or take those images and make running containers using other types of commands, and this is all done through the Docker Client, and again, this works with Docker Toolbox and with Docker Desktop. As mentioned, this tool, Docker Client, is going to allow us to interact with the Docker Engine, the Docker daemon that's running behind the scenes. Through using this tool, we can build and manage images. We can then take those images and run and manage containers. So let's look at some of the commands that you could use as you start to work with the Docker Client. Now, some of the key commands are going to be shown here, and this is a small subset of the commands. There are quite a few that you can run. One of the big ones you'll use, though, is called docker pull. You might find a Node.js image or a ASP.NET or PHP or whatever it may be, you might have an image up in Docker Hub and you want to pull that from Docker Hub down to your development environment. Well we can use the Docker pull command to do that. Once we have an image we can run it. We can use docker run to do that. We simply say docker run and then give it the name of the image that we want to run. We can also list all of our images by simply running docker images. And then when it comes to containers, we can run docker ps, and this will list all of the different containers that we

might have available. Now, once you have the containers and the images and everything available, we can then start containers, we can stop containers, and do all kinds of things that we're going to look at throughout the course. And so before we go too far, though, let's take a look at using the Docker Client with these commands and see how they can interact with our different images and containers that we may have or that we might want to grab from Docker Hub.

Docker Client in Action (Mac)

Let's take a look at some of the commands we can issue. So the first thing I'm going to do is just type the word docker. And if I just hit enter or return here, this is going to show all the different commands. And you'll notice there's quite a few commands. Now we're only going to focus on just the essentials that we need to get started, but those are the command you can use. And there's a lot you can do with images and containers and a bunch of other stuff as well. Now from here, let's learn how we can pull an image off of Docker Hub. And we talked about Docker Hub a little bit earlier, and if you go to hub.docker.com, you can get to it. Now you can certainly log in and create an account and all that. But if we just come up to the search, I'm going to type hello-world, and this'll pull up the hello-world image. And we'll click on it, and you'll notice there's this Docker client pull command that we can issue. So I can actually just copy that from here, get information if I'd like to read up on it. But we'll go on back, and I can just paste this in. So let me clear this and we'll just paste docker pull and the name of the image. Now this is going to pull down the layered file system, it's called This is the actual image itself, very small, you can see. And so now I have this image locally. But how do we know if it's really there? Did it work? Well, the second command we're going to look at for Docker client is, we can type the Docker client tool again, and now I can say images. And what this'll do is list all the images that I have installed. And it looks like we now have one image. Let me make this just a little bit bigger. You can see it's hello-world, we have the latest, it assigns a unique ID to it, and it looks like it was created 11 weeks ago, and it's very small, 960 bytes. Now that's how easy it is to first off pull an image off of Docker Hub and then actually see what images do we actually have? Now I mentioned a couple times, an image on its own is not ultimately that useful because we need to take that image and actually get a running container. And so what we can do from here is, we can say docker run and then we can give the name of the image. We're going to do hello-world. And this will now run the hello-world image as a container, and you can see all of this output. And if you see this Hello Docker, then you did good. It worked. And so we now have a hello-world container. So we have an image that's sitting there. We've now taken that image and made an instance of the container that's now, it ran and then it actually stopped. And so how do you know what containers that you have available? So I can come in and type docker ps, but let's see what we get here. You'll notice nothing shows up, which I know the first time I saw this, was a little bit confusing because I knew I had a container because that's what the run command that we looked up here does. So what's going on? Well, docker ps, this command, only shows the running containers. So how do we see all containers? Well, we do -a, and that will tell the Docker client ps command, I would like to list all containers. And there we go. So now we have the container ID, we have the image that was used for it, we have, it's kind of wrapping here, this is a sort of friendly name it comes up with if you don't want to refer to this guy, we created about a minute ago, and the status is it's Exited. So this particular container right now is not up and running because this container just outputs this log data you see up

here, and then it kind of stops. So that's a really important little add-on to the ps command for a command line switch to do docker ps -a. Because again, if you don't do that, you're not going to see it in this case. Now, that's not that useful of an image or a container, so how do we get rid of these now? That's great, we see it works, but now I'll probably never, ever use it again. So what I'm going to do is, I'm going to come in and say remove and that's remove container. And then we have the container ID right here. And then we also have a little alias, but I just normally type the first few characters. So I'm going to do 59f. So we'll do 59f, I'm going to hit Enter. Now When I do docker ps -a, you'll notice I don't have any containers left, so we deleted it. Now what about the images? Let me get do a clear here. Well, let's do docker images again. Okay, It's still there. We deleted the container, but we didn't get rid of the image. So it's really similar to what we just did. Its Docker rmi, and then, again, we can take the image ID and just do the first few. So in this one sense there's only one. I'm just going to do 0a. And there we go. It just deleted the layered file system for that particular image. So that's an example of how easy it is to use the Docker client command line tools to get started with pulling an image, viewing images, and then taking those images and converting them into running containers. So now that we've seen the basic commands for Docker client, let's pull down the Nginx image that I showed, if you watched the Kitematic demo earlier in the course, and see how we can get that actually running as well. So if we go back up to Docker Hub and go to hello-world and I'll just kind of research on this. We'll hit Enter there. And you'll notice that there's quite a bit of things we can do. There's a tutum hello-world. There's also a Kitematic one down here. It looks like I'm not finding it immediately, so we could actually search for Kitematic. And there it is right there, so let's click on that. And just to save a little bit of typing, I'm going to go ahead and grab this pull command and just paste this again into my Docker client terminal that we have here. So we'll pace that in, and now this image will have a little bit more to it. It's bigger than the last one, so it'll take a moment to download but It's pretty quick, and there we go. We have it. So there's docker pull again. Now we can do docker images. There it is. So there's our Kitematic, it's the latest, there's the unique ID it gives it, and it's about 6 months old, it looks like. So now we can actually start this image up and get it running. And to do that we can, again, do docker run and then give the actual name over here that we have of the image. So I'm just going to copy that and paste it in. Now this particular image, though, has a port that we need to set. And so you can kind of think of it this way. The image is going to become a running container. Now ultimately what we have is a machine that hosts the container. Well, the machine, as a port we're going to hit, because you'll notice I already have an IP address up here typed for the machine. But we can also set the port the we're going to call on that machine. Now when that gets called, it's then going to call into the appropriate container, in this case, the Nginx container, which is a reverse proxy type of tool. We can also set a port inside of the Nginx. Now, normally Nginx is kind of a front-end server that'll serve up static files and forward more complex requests to back-end servers, ASP.NET, Node.js, and others. And so normally, it's on port 80, if it's a kind of a public-facing website. So what I'm going to do is come into here, and we're going to use a command line switch on run. And I'm going to say that I would like to run this image but I want to run it on port 80 for the machine, and that's going to forward internally to port 80 in the container itself. Now, this is a really important one because we need to set what is the port we're going to call on our actual machine and then what's it going to call on this container that's going to get created based on this image? So let me go ahead and just hit return here, Enter, and you'll notice this now started up my Nginx container, in this case. It converted from the image into the container. So I'm going to come up here, and now if I hit the IP address for my machine, we should see an

Nginx output here, and there we go. Now, this looks very, very similar to what we saw earlier, if you watched the Kitematic demo, again, because it's the same exact image. It's just that we're now using the terminal here to actually work with this particular image and container. What I'm going to do from here is, I can actually just start up a new kind of Bash tab here and I'm going to go into the docker ps command that I showed you earlier. And you'll notice that right off the bat, because I didn't click on a Quickstart terminal, I get this Cannot connect to Docker daemon. Now, one way around this is, I can close it and just open up a different Quickstart. But let's go ahead and use what we learned earlier. I'm going to do docker-machine env, default is my machine, and then I'm going to run this eval command. Okay, so now let me try docker ps. All right, now it works. And you'll notice this command terminal is now tied to my default machine. And again, I didn't have to do that. I could've just clicked on the Docker Quickstart terminal here. But that's a nice little thing. So here we go. We have the container ID, here's the image it's based on, there's a little start.sh script that's run, we started about a minute ago, and it looks like the status is up. It's wrapping a little bit there, but you'll notice status is up for about a minute or so, it looks like. Now, we can come into here and we can say docker stop, and then we can actually list just a few of the digits here. So I can just say docker stop 109. And now this is going to go in and try to stop that particular running container that we have going over in this tab right here. And so we'll let this run just for a sec. Okay, there we go. Now let's do docker ps, and if it's stopped, it shouldn't show it. All right, and it's empty. So now we'll do docker ps -a, and there it is, but you'll notice that the status is Exited. So that's how easy it is to pull that down, and then you now know how we can get rid of this as well. Once it's stopped we can say docker remove and then give it that same container ID. So we'll just do 109, since that's a quick and easy way. Okay, so now docker psa. It's gone. Let's go to docker images, and there's the image. Notice the, again, image ID here. So we can say docker remove image 385, and that now deleted all the parts of the layered file system. And there we go. So now our container's gone and our images are gone. And from a development standpoint, this is pretty awesome, because now my machine is completely clean of this Nginx server. I don't have to worry about other files sticking around because I got rid of the actual image and the container. Now compare that to manually installing different servers and databases and things like that, and I think you'll find that that's a pretty compelling thing we can do in the development environment. Because I don't know about you, I tend to like to keep my machine pretty clean. So that's an example of some of these different Docker client commands that we can actually run in a Mac environment.

Docker Client in Action (Windows)

So, the first thing I'm going to do is I'm going to run off to hub.docker.com, and I've already typed in this hello-world image that I'd like to find that's up there up in the cloud on Docker Hub. And so let's go ahead and find this, and you'll see the official hello-world image, and this is a very basic image you can use to get started. So if we scroll on down, you'll see a description, some information about it, you'll see some example output of what we would expect if we run it as a container, and then over here to the right you'll notice that I can run this command that's a docker client command called pull. Really simple to run, you simply say docker pull, and the name of the image. So I'm going to copy that, run on back here, and let's paste that into our terminal window. Hit Return or Enter, and this is going to pull down a layered file system. And you'll see the pull is now complete, very fast because it's a very

small image, and now we're kind of ready to go. So it pulled that image down to our local machine. Now how do I know that it actually worked? Well, we can come in and say docker images, and this will list all the images that we have on the machine. And it looks right now that I have this hello-world, it's the latest, here's a unique IMAGE ID it assigns per image, it was created about 12 weeks ago, and it's really small, 960 bytes it looks like. Now from here we have an image, but images on their own aren't really that useful. They're like having a blueprint, but never creating a building. We want to create the building. We want to create the container that can do something. So now I can use the docker client command called run, and I can say docker run, the name of the image that you'll see here, hello-world, hit Enter there, and there we go, this is the actual container running. And so you can see Hello from Docker, this message shows that apparently we're working, so we've done pretty good so far, we have a really, really simple image running, and they have some other info you can check out there if you'd like. That's not super impressive, obviously, but we do have a container. Now is that container still running, or what happened there? So we can actually see all the running containers by doing docker ps. And so I'm going to hit Enter there, and you'll notice it's empty, which is a little bit weird, because, you know, I do have a container, obviously it ran, but it must not be running. So if we want to list all the containers on the system, we can say docker ps -a, and that'll show all of them. So we'll hit Return there, and there we go. So this is wrapping a little bit, so I'll make it a tad bit bigger. But you'll notice that we have a CONTAINER ID, and that's assigned per container. It's based on the hello-world image. There's a command it runs internally, just hello. We created it about a minute ago, and it exited about 55 seconds ago. Now it also gives it a little more friendly alias, if you will. And this particular alias is kind of something you can use instead of the alphanumeric characters you can see over here for the CONTAINER ID. Alright, so we've now run the container, we can see the container, but it exited. So this is a different container, this isn't one that you run the container and it stays up and running like a server. It just runs and then it just shuts down, so it's a very simple hello world type of example. Alright, so let's get rid of this container then. We know it works, but we really don't need it anymore, and you'll probably never, ever use it again. So we're going to do another docker client command called remove, and this removes containers. Now, I'm going to go ahead and use the CONTAINER ID, but I really don't want to type all this. I know when I first started using Docker, I didn't realize that you don't have to type the entire CONTAINER ID, so I went in type the whole thing, though, but you don't need to. We can actually, in this case we only have one, so I could get away with 24, I can get away with 2 if I wanted, but let's go ahead and do that. And you'll see it echoed back out the container it removed. Now let's make sure it worked. We'll do docker ps -a again, and everything's gone you'll notice. Okay, so the container is gone now. Now what about the image? Well, the image is still there, and I probably don't need that on my system, so let's clean that off. And we can do that, and remove it by doing docker rmi, remove image. And then just like we did with the CONTAINER ID, there's an IMAGE ID here. So we only have one, so it's pretty simple, I'll just do like 0a, and now it just deleted that layered file system. Now, if we go back and do docker images, you should see that it's completely gone. So now we've downloaded the image, or pulled the image, we've run it, the container immediately stopped. We removed the container with the rm command, and now we just removed the image, so now there's really no trace of this on our system, and that's a great feature that we're going to talk more about in a moment with Docker in the development environment. So that's an example of how to get started with those commands. Now let's take a look at how we can pull a more robust image from Docker Hub and get that up and running on our machine. So if we go back over to the Docker Hub site, I can come in and search for a

hello-world, but for the nginx. And if you saw the Kitematic demo earlier in the course, I'm going to do the same thing, but we're actually going to do it using the docker client tool. So I'm going to come in and we'll just search for kitematic, and we can just do hello-world here. It should pull it up, and there it is. So we can view some information about it. There's not a whole lot on this one, but it's a simple nginx reverse proxy container. And you'll notice over here, again, just like with the hello-world image, I can also pull the kitematic/hello-world-nginx image. So let me make sure I grab that whole command, and I'm just going to come on back and paste this in. So we'll paste in the docker pull command, and this one will have a little bit more, so this is going to pull down, again, the layered file system. You'll see this'll start to fill in. It's still pretty fast. Alright, so we're ready to go. So I'm going to do docker images, and there we go. We have the kitematic/hello-world-nginx, latest, there's the IMAGE ID again, and we can see the age and how big it is. So this one's a little bit bigger, looks like about 8 MG or so. So the next thing I'm going to do is we have the image, and just like with the hello-world image, I want to go ahead and run this. So we would do the same thing. We would say docker run, and then we would put the image name. Now, because this is an actual server, it doesn't just write simple log output, there's a little bit more that we need to supply here. Now, we have a Docker machine, in fact, that machine IP is shown right up here because I'm going to use it in just a moment. We saw that when I started the Quickstart terminal. And that machine needs to be told what port do you want to call to come into on the machine, and then we have to tell the machine, okay, well, once you get on that port, we're going to do port 80, how does it call into the container, and what port does the container actually have? And I like to think of it as a bubble around the container, and on the outside of the bubble is the machine port, and then it's going to call a port that's the actual nginx port that's in the bubble or in the container, in other words. So let me show you how this works so it'll make a little more sense. So I'm going to say -p, and then we're going to do 80, and that's going to be the port for the machine, colon, and then that's going to say I want to forward from port 80 on the machine to port 80 in the container itself. Now, if we wanted to do maybe 5000 on the actual machine, but 80 on nginx we could do that, but nginx is typically used as a frontend type of reverse proxy server, it can serve up static files and then forward requests to more complex backend servers like ASP.NET, and Node.js, and PHP, and things like that, so we're going to do 80:80 and this will forward it. And we're going to say docker run, on this port on the machine and on the container, and then we have to put the name, so it's going to be kitematic/hello-world-nginx. Alright, so now that we have that in place, we can go ahead and start up this nginx server. So we'll go ahead and run that, and there we go, it looks like that container is now up and running. So what I'm going to do is leave that up, and I'm going to right-click here on our Quickstart terminal, and I'm going to start a new terminal. And that'll link us up to the default machine again, and there's the IP address. And now let's see, what do we have as far as containers. So we're going to come in and we're going to run the docker ps command, and it looks like we do indeed have a container, and you can see that it is up for about 23 seconds, up and running. There's the IMAGE it's based on, there's the CONTAINER ID, and then here's the port forwarding I was talking about. So the IP here just is kind of generic, but this will be the machine IP, :80, that forwards to port 80 in the actual container itself. Now that container is up and running, as you can see here. And it looks like when we started it up, we have this start.sh that the nginx image actually had in it, and that actually started up the nginx server. So I'm going to run over to this tab now, and we're going to try to refresh here. We're on port 80, so obviously I don't have to put, you know, :80. We could, but I'm just going to hit Enter, and there we go. So we now have on our development machine an nginx container up and running. Very cool, because I didn't officially install anything from the

nginx site, we're just using, obviously, Docker images and containers here. So I'm going to run on back. Let's bring both of these back up. You'll notice that a request was made here. It shows the GET request was made, it was a successful 200, and some other information about the browser. And then if we come on back here, you can see that we're back where we were, and we can see that it's up, and all that. So, now what I'm going to do is let's go ahead and try to stop the container. So we'll go ahead, and not to confuse terminals here, let's go ahead and leave this one up. And I'm going to type docker stop, and then I'm going to take the CONTAINER ID over here. Now, again, we don't have to type the whole thing. In this case, I could say, for instance, d7, and that's going to stop, using the docker client tool, that particular running container. And this will take just a moment for it to stop. Alright, so once it echoes back out the ID you typed, that pretty much means it stopped. So if we type docker ps, we shouldn't see it, and we don't, but if I do docker ps -a, for all, we should see it. But you'll notice that now it says it's Exited, about 14 seconds ago. Alright, great, so we now have a running container that we stopped. Now we've seen that because it stopped we have to do the -a switch again to see it. And now let's go ahead and clean it up. Now, this is one of the more exciting features, I think, from a development standpoint. Instead of installing a server on your physical machine, whatever it is, database server, web server, normally when you uninstall it, you know, it seems like there's always a few files left over. But in this case, because we're using images and containers, we can use our normal docker client commands, and I can say remove, give it that d7 container, and now if we do docker ps -a, you'll notice everything is gone from the container. So, alright, that's great, but what about the image? Let's do docker images. We still have the image. Now, normally, if you're going to be reusing this image to make other containers, you'd probably just leave it if we do it a lot, but in this case, let's just say, hey, I'm done with it. I really don't want it on my machine anymore. I've maybe tested something, and everything is working great. Well, just like we did earlier, we can do remove image, and in this case the IMAGE ID has this 385. So we'll go ahead and do that. That completely deletes that image, and now if we do docker images, you can see we're clear, and docker ps -a, of course, we're clear on containers. And this is pretty cool I think. I'm a little bit picky on my development machines, and I like to keep everything really clean, and so when I'm done with something, I really would like all traces of it to be removed, and now it is. And I think this is a very, very cool feature for development that I literally can get, whether it's a database server, a server that's a reverse proxy like nginx or others, up and running quickly on my machine without a lot of effort, just a few commands, and then I can completely remove all traces of it using these docker client commands. So that's an example of some of the different docker client commands that you can run on your Windows machine.

Docker Commands Review

Let's go ahead and do a quick review of some of the key commands so we cement our knowledge and make it easier to remember them. So some of the image commands we talked about were docker poll. We talked about docker images lists all the images you have on your machine. And then we talked about how you can remove an image with docker rmi, and you give it the image ID that's available when you run docker images. Now when it comes to container commands, we talked about docker run, and the image name. And docker run, interestingly enough, will actually pull the image if it's not already found locally. So although technically you run docker poll first and then run docker run, if you want to save a step, you

could actually run docker run and then give it the image name. And if it doesn't find it, it'll go download it and then we'll run that container. We also talked about how you can list the containers. Now if you only want the running container, so you could do docker ps. But if you want to see all of them, you would add -a to the end of that. And finally, we talked about docker rm, which can be used to remove a container. So that's an example of some of the key commands that we've covered throughout this module, and I hope that helps cement them a little better in your mind. We'll be using them throughout the rest of the course, actually.

Summary

To wrap up this module, we've talked about several different Docker Client commands that you can run, and we also talked about a special command that's just for Docker Toolbox, called Docker Machine. So, as a quick review, we talked about how Docker Machine is used to actually hook up to that running VM that's running in VirtualBox, and that's a Linux VM that we need to hook into. We talked about some of the commands you can run there, but again, you're only going to need that if you're using Docker Toolbox. If you're on Docker desktop, you don't have to worry about it because they kind of hide all that from you. Now the main one that we all need to know, regardless if you're on Docker Toolbox or Docker Desktop, is the Docker Client commands, so we talked about commands such as docker images to list all the images, docker rmi to remove an image, and then we talked about several container commands as well, the most important being docker run or maybe docker ps, and docker ps shows our running containers and then, of course, we saw how docker ps-a can show all the containers, even stop containers, paused, those types of things. So knowing these commands is a really good start to interacting with pulling images and then converting those images into running containers, and now what we're going to do is keep building upon this knowledge and adding more and more and talk about some other commands and some other features of Docker that we need to know to work with it. So let's go ahead and jump on into the next module.

Hooking Your Source Code into a Container

Introduction

We've learned how to work with images and containers in Docker, but we haven't seen how to hook our source code into a container, so that's going to be the focus of this module. Now we're going to start off by introducing something called the layered file system, and this plays a really critical role with your images and any running containers that you have. So as you want to, for instance, write to a log file or have database files or even work with source code, it's important to understand how Docker actually works with files. Now, once we talk about that, I'll introduce a term called volumes. Volumes are really important, especially as you work with your source code, if you want to get that source code hooked into a running container, so I'll introduce it here. And then we'll talk about Docker client commands that you can use to actually create a volume. Now, from there, I'm going to show you some actual examples of hooking real source code into running volumes, and I'll show all the tools that even create the source code from scratch, get that up into a volume that's associated with a

running container, and how everything works there. And then once we're done with those demonstrations, I'll show you how we can, with just a really simple command, remove a volume that might be associated with a running container. So the big question that we're going to answer in this module is, how do you get your source code into a container? Because that's really what we're after here. And it turns out there's actually multiple answers. We're going to focus on this first one. How do you create a container volume that points to your source code? And that's what I'll address and show you how to do. Now later in the course, I'm also going to show you how you can add your source code into a custom image that can then be used to create a running container, and I'll show the tools and how all that works as well. But for now, we're going to focus on container volumes, and I'm going to show you how we can get started using those and how the file system works.

The Layered File System

Before we can talk about how we can get our source code into an image or a container in Docker, we first need to understand how Docker images and containers work and discuss something called the layered file system. Now I've mentioned this term a few times earlier in the course, but we really haven't gone into any good details on what it is and the role of plays with our images and containers, so let's talk about that now. Now, from a high level, a dessert perspective in this case, we have a bunch of layers here you can see. And at the very bottom we have the base layer, and then we add layers on top of that and build up and up and up until we get the final dessert in this case. Now you may immediately say, what does that have to do with Docker images and containers? And actually, the concepts have a lot to do with images and containers, Docker images and containers are actually built of this layered file system, and so you can think of instead of the dessert layers, layers of files that build upon each other. And you're going to see that's good for a lot of reasons. It's good for disk space, it's good for reuse, and even other things. So let's take a look at an image and see how these layers play into that image. So here's an example of Ubuntu, and let's say that we grab this off of Docker Hub. And so we've got all these different layers, and this is our layered file system in the image. Now the file system and the layers that compose it within a given image, they're all read only. And so once that image is baked, you're not going to be writing anything to that image from a container, for instance. The image has the files, they're kind of hard-coded in the image, and they're ready to go and be used, but you can't actually write to this. Now that may seem a little bit limiting at first glance because we might have images with a database that needs to write files. Maybe we have to log some files. Maybe we have some source code we want to swap and change as a container is running. So fortunately, while images and the file system they have is read only, a container builds on top of this and gets its own thin read/write layer. And really, that's the main distinguishing factor between a container and an image. An image is a set of read only layers, whereas a container has a thin read/write layer. Now, as you write to that layer, if that container gets deleted, then the writable layer also gets deleted. But coming up, I'm going to show you how we can change that and use something called volumes. But for now, just understand that it is possible to write to a container and do log files or database files or even have source code that does something like that. And we need to put it, though, either in the image as a baked in layer or up in this thin read/write layer of a container. And we're going to focus first in this module on the container layer and what we can do there and how we can use it. Now, as mentioned, these file layers that we're using within our images are really, really efficient

when it comes to disk space and reusing things like that. So, as an example, if I were to use this Ubuntu image and make a bunch of containers, then all these image layers that you see here, and they all have unique identifiers per layer, you'll notice they have some universal, unique identifier, and these are all going to be shared across all the different containers. So that's really, really good for disk space because we don't have to make a copy of that entire file system. And that's why it's pretty quick to actually pull down different images, especially once you already have some images installed. Because if you take this one on the very bottom of Ubuntu that starts with d3, if that particular file layer is used in other images, then it'll just detect that, hey, I already have that, and it won't have to redownload it. It'll just share it between those images. Now, in the case of each container that's created here, you can see they all have their own unique read/write layer. And so that's going to be okay. Each one can uniquely log or store database files. But as mentioned, if you delete the container, you also get rid of that thin read/write layer, and that's where some things called volumes are going to come into play. Now that we've talked about this layered file system, this can help us answer the question we addressed earlier in the module of how do you get your source code into a container? Well, at a minimum, you could put it in the image, and we'll talk about that in a later module, and I'll show you how to do that. But as mentioned for this particular module, we're going to focus on the container level, that thin readable/writable type of layer, and we're going to integrate the source code into that container using that particular layer. So let's go ahead and take a look at more information on how we can do that with containers in something called volumes.

Containers and Volumes

Up to this point, we've learned about the layered file system and how it works with images and containers, how containers are a little bit unique and have their own thin read/write type of layered file system, and we call that the container layer typically. Now I mentioned though that any changes made while a container is running that are written to the writable layer, they kind of go away if a container is removed. So if you delete that container, you're also going to delete the file layer that is the read/write layer. Now obviously in scenarios when you have database files and logs and source code, we might want to keep that around potentially, especially while we're doing development and just trying to use Docker as a development environment. So fortunately, Docker and containers have another feature we can use called volumes. And what I'm going to do in this section is just introduce volumes, and then later we're going to learn about how we can use those volumes. So what is a volume? Well, a volume is nothing more than a special type of directory that's associated with a container, and typically you'll hear it referred to as a data volume, and that's because we can store all types of data. It could be code, could be log files, could be data files, and more. Now we can share and reuse these among containers, so it is possible for multiple containers to write to this volume, or you could just have a single container that has one or more volumes that it writes to. And what's nice about this is any updates to an image aren't going to affect a data volume. It stays separate. Also, data volumes are persistent. So even if a container is deleted and it's completely blown away from the machine, the data volume can do still stick around, and you have control over that. Now, from a high level, you can think of volumes this way. If we have a container, then we can come in and define a volume within that container. So in this example, /var/www, where do we want that to write? Well, you kind of have two options. You can let Docker figure it out, or you can give it your own. And I'm

going to show you how to do your own custom volume coming up in the next section. But for now, let's just know that when you write to a volume, so let's say that your code in the Docker container actually does a write operation to this /var/www path, well, that is really just going to be an alias for a mounted folder that is in your Docker host. Now remember that the Docker host is actually hosting the container. So if you're running on a Linux system or a Windows Server 2016 or higher type of system, then the host would be that OS. It's the thing that the container is actually running on top of. And so in this example, if we had a volume that we wrote to, instead of writing into that thin read/write type of layer that is associated with the container that we talked about with the layered file system, it can actually write it up into this mounted folder area that's part of the Docker host. Now if you delete the container, the folder that's on your Docker host, it can actually stick around, and you can preserve all of that code if you'd like. So that's a quick introduction to what a data volume is and what a volume in general is in the world of Docker containers. So now what we're going to look at is how can we actually get our source code into our containers using volumes? And we'll see how we can set that up using things like the Docker client.

Source Code, Volumes, and Containers

Up to this point, you've learned about the layered file system and how it's used with images and containers, and we've also learned about the basics of volumes. But let's go a little more in depth into volumes and how we can actually use these to store some source code. So earlier, we looked at containers and saw that we can define a volume in a container. Now, we haven't quite seen the syntax to do that, but I'm going to show you that here. And I mentioned that when you write to a volume, if you set that up, that it's actually going to write to some special area that's on your Docker host. And by default, Docker takes care of that. It takes care of creating this area where it mounts this folder. And so I like to think of the var/www/volume that's in the container here as really being an alias that points over to the Docker host and this mounted folder type of area, and that's where you can put your log files and that type of stuff. Now to do this, we normally run a command like the following to start up an image and make a container. So we can say Docker run, give it our port, and we have the external and the internal container port, and in this case, I'm going to run the Node image. Well, if we actually want to have a special area, a data volume, where that Node app could write to, then we can change it to look like this. So I can put -v, and that stands for volume, and in this case, say, /var/www and then put the image name. Now the var/www, or www, however you like to say it, that would be the volume. And then the area that it writes to would be in the Docker host, and so it would kind of look like this. We create a volume, this is the container volume alias, but it actually is going to right to the host area, and Docker again will automatically create that. Now, where does it store it? How would you know? Let's say that, for instance, your Node application writes a log file out to this var/www folder; how does it know where that's going to be? Well, what's going to happen is, Docker kind of magically makes that mounted folder. And the way you can find where it is is by running docker inspect and then the name of your container. And so we could come in and do that, and if you scroll through the information it gives you, you'll see a Mounts area. You can see that over here. And the Mounts area has a name, and it's going to be a really long, unique identifier, and then a source path, and it's also a fairly long path. You'll notice it's in this Mount, mnt, folder, and that's going to be on your host, your Docker host. The destination that the container actually writes to is going to be this var/www. And so we have the host location

defined there in the source, and then we have the volume location that's in the container defined by the destination property that you can see. And so, in this case, Docker is automatically taking care of where that data gets written to. But you now have to know if you ever wanted to go get it outside of the container, have to know how to get to this path, which can be a little bit long. And it works great in scenarios where you don't want to control it. You just want to set up a container volume, write to it, and then have Docker take care of storing that somewhere and persisting that data. And this is the default way it will do it. Now, the other option is, we can actually customize our volumes. Because in this case, Docker is determining the mount location, the folder, where the var/www is actually going to write to. So let's look at how we could actually customize this. So instead of actually having Docker set up the folder that it writes to on the host, we could come in and give it our own folder path. And in this case, I'm just saying /source, but it could be a variety of paths. And this could be in your source code, could be where you want your log files, your database files, or whatever it may be. So this gives us an option to work with, for instance, source code in this example, store that in a certain folder, maybe on your Mac or your Windows machine, and then have the volume actually read and write to that specific area, which in this case, would be our /source. So what does the Docker client command like like to make this possible? Well, again, if we start out with the following where we just run the node image, we could change it to this type of pattern. We could say -v, and that again creates a volume. This $(pwd) basically says, hey, go from the current working directory and use that as the host mount. In other words, use that as the folder where I want to put my source code. Now the actual container volume, though, would be this /var/www, and then, of course, we have the name of the image, in this case node. So what this will do is create a volume in the container, which is going to be var/www, but when you write to that or when source code's read from that as the Node container's actually running, that's it's actually going to look in the host location, which would be the current working directory. So if you set up a /source folder and that was where you ran the command prompt from, then that would be your current working directory. If you were in your user folder, then that would be the current working directory. Just depends on where you are. Now, if you do an inspect on this, things change a little bit, and so we'll run docker inspect on the name of your container, and here's what we'll see in the Mounts area. So again, we'll always have a name, which is a unique identifier, but you'll notice now that the source on the host location is what we wanted, in this case, /source. Now, if you're on Mac or Windows, Docker's smart enough to allow you to work with source code directly on your Mac or your Windows machine, have that talk through VirtualBox and up to the container, and it kind of does some magic there to make all that happen. So that's really nice for us as developers because now I can work with my source code right on my Mac, Windows, or Linux machine but have my container be loaded up and then reading the source code or writing to that area using this volume support. Now the destination, or the volume in the container, the destination as far as the container is concerned, is now going to be var/www again, so that part doesn't really change. But again, that's kind of like an alias, is how I like to think of it, and that's actually going to read and write up to this /source. And again, anytime you're working with this -v syntax as you run a particular image and make a container, whatever you write to that volume gets persisted. So even if you delete the container, that's going to stick around, and in this case, that's a good thing. Obviously, if we delete a container, we don't want it to delete our source code. So that's an example of how we can get started using the Docker client with setting up a volume that could either automatically generate a folder using Docker on the host or how we can control it by using

our own syntax. So now that I've shown you the basics there, let's look at some examples of how we can actually hook source code into different types of containers.

Hooking a Volume to Node.js Source Code

In this section we're going to take a look at how we can get Node.js source code into a running Docker container by using volumes. Now, this source code is actually going to live on our local development machine, but we're going to magically link it into the container using volume support. Let's go ahead and get started here. So I'm going to come on in and start up the Docker client. I'm going to use something called Express. This is a web framework for Node.js, and it has a little feature called express-generator that will generate a little sample site that will make it easy to get started with some Node.js code. So the first thing I'm going to do is run an npm command, and this will install some modules. We're going to install Express and express-generator, we're going to do these globally because express-generator will add some command line integration support, which you'll see in just a moment. All right, so we're kind of ready to go, and what I'm going to do from the location, which is really the user account I'm in right now, is I'm going to create a new folder and generate some source code in that folder that's for Node.js. And the way we can do that is run express, I'm going to give it a folder name, we'll say ExpressSite, and then I can give it a technology for how to render the views, I'm going to use something called Handlebars. So that will run, and now what we have to do to get this site up and running is first off, run these commands. So we're going to cd into the folder, and then we're going to install all the dependencies of this particular web application. So this will take just a moment to pull these dependencies down. All right, we're ready to go, and then I'm going to get this running. Now I do have Node.js running on this local machine of course, that's how I'm running npm and these other commands, but the goal is not to run node here, it's to run it in a Docker container. But this will show us that the source code is running properly, at least locally to start. So, now that I'm in the ExpressSite I can just type npm start, and this will fire off this little web server here, and now we can come on in and there we go, we now have a little Express site. All right, so we're off and running. Now that's nice and all, but that's actually just running Express directly from this particular folder. So I'm just going to go ahead and kill that, we'll leave this open to start. So the next thing I'm going to do is show you how we can work with volumes. So we're going to come back to the Docker client, and if I run docker images, I've already pulled the latest Node.js image that we're going to ultimately create into a container from Docker Hub, and so normally to run this we would just say docker run, we'd give it a port, we'd give it the external port, I'm going to say 8080, and then you saw that the Express website that I got going is actually going to run on 3000. And we would say node, but if we run this, we don't have any source code. So, basically what will happen is the container will try to start, it will see there's no command to run, and it'll just exit. It'll just stop. So we're not going to do that. What I'm going to do first is create a volume. Now, the first volume I'm going to show you is going to be this var/www, or I like to call it dub dub dub to shorten it up, but this is going to create a volume, but it doesn't actually point to our source code. This is just an area that if we did have something running in the container, and the path for the running app wanted to write to var/www or read from it, then it would create a volume outside of the container in the host machine. And so now I could say node here for the image and we'll go ahead and return, and now it just started the container, but again there was nothing to run. So if we do docker ps -a, you should see that it's exited. So really, what happened is it

tried to start it, didn't see anything fancy, and then stopped it, but we should now have a volume under the covers. Now you'll notice the container ID. Now what we can do from here is we can say docker inspect, and then give it the start of that container ID, or we could do the drunk_borg, that's one of the more interesting ones I've seen as an alias, but we'll go with the 03 here, and this will spit out a whole bunch of information about that particular container, and I'm going to scroll back up and we're going to look for something called Mounts. All right, there we go. So there's our Mounts you can see right there. Now you'll notice that we have a source on the host system that's a really long path, and it uses this name, which is a unique identifier for the particular volume, and it kind of buries it in this folder structure that you'll see right here. Now on the container itself, the alias for this path that's on the host is just var/www. So again, if we read from var/www or we write to var/www, what it's going to really do is be writing to this location or reading from this location. Now that doesn't help us as much with source code because I'd have to get my source code into that folder, and that's not a path I really want to work with, so what I'm going to do is come back down and we're going to go ahead and remove the container. So again, if we do docker ps -a for all, this will show us all the containers, even the exited ones, and then I'm going to do the normal docker remove that we've learned up to this point, but I'm going to add one more thing. I want to make sure that that volume that's on the host machine also gets removed when we removed this container. So I can do the 03 for the container ID, but by adding -v that'll go ahead and clean up the volume. Now normally when you remove a container, it's not going to delete the volumes because there might be another container that's using that volume. So if this is the last container that uses that volume, you typically want to clean that up, and that's what I'm going to do here. All right, so now if I do docker ps -a again, you'll notice that we're all cleaned up and we're good to go here. All right, now we have that ExpressSite folder that I created earlier though, so what I want to do is let's cd into that, alright, and we can do an ls to basically list everything, kind of like a dir that you normally do on the Windows side, and you'll notice that we have this app.js and some node_modules and package.json, and this is pretty standard folders for an Express site. So what I want to do is link this folder into the container and then start up Node, very similar to what I just did earlier. So what we can do is use the volume support that I talked about in a previous section in this module. And so we could do something very similar, we can say docker run, give it a port on the external 8080, internal it's 3000, but this time when I create the volume I'm going to say let's start from the current working directory, and this is the little shortcut I showed earlier you can do. And this is going to be the directory that the volume in the container is actually going to point to. Perfect, because that's what we want. We want to point to our source code, which is in this Express site. Now, the name again we're going to use is var/www. Now I made that up, it could have been something else, but I'm going to go with that. And then normally we would say node, and then if you want to run any commands in the container, we could run that npm start command. Now we're going to have a problem here. It won't actually run npm start from this folder. It'll run it from a different location. So I'm going to show you a little trick you can do here called the working directory. So we're going to say -w, and that stands for the working directory, it's a shortcut of what is the startup directory, what's the folder in the container where any command should actually be executed from. So it kind of sets the context of where to run these commands. And I'm just going to say /var/www here, put the image, and then after that we can put the command that we want to run, and I want to run npm start. Alright so to review, we're going to say, hey Docker, let's run on port 8080 on the external, 3000 is going to be internally what we're going to run and I picked that on purpose because that's what this Express site will use by default. We could change it, but that's how it's set up

currently. We're going to set up a volume that points to our source code in the current working directory, and then the volume, though, that's inside of the container that's going to point to this ExpressSite folder is going to be var/www, then we're going to go ahead and use that volume as our working directory. That way when I run npm start, really what it's going to do is forward the call from var/www into this ExpressSite, which will call into an area over here. Alright, so a lot going on, but let's go ahead and try it out, and I'll just hit Return here. All right, now you'll notice it started it up, but this time it's not running on my local machine, it's actually running in the Docker Node container. So this is very, very cool because I've now linked my source code into this container, and even if I didn't have Node installed, if I just had the source code, but didn't have Node on my dev machine at all, then we could still work with Node because obviously it would be loaded up in the container. All right, so let's try this out. Now, instead of going to localhost, I want to go to the IP address. Now, this will be the Docker machine IP, so we're going to go to 192.168.99.100. I'm not going to go to 3000 though. Back here we said that, what we'd put, we want to go from 8080 to 3000 internally in the container, so I want to use 8080 right there. We'll hit Enter, all right, and you'll notice we get the exact same Express site. Now, just to kind of prove this, let me dive into the source code real quick, and so I'm just going to run off to this folder in Users, and where is Express, there it is, ExpressSite, and let's just come in and I'm going to do a change to the view. And right now it's loading the home page, this is called index, I'm just going to open this up in VS Code editor. We'll do a very simple edit. So it says Welcome to title, and say Welcome to title running within a Docker Container with a volume, no less, which is pretty cool. Alright, we're going to go ahead and save that, that should now be committed in the source code, and you'll notice, look at that. So now our source code is linked, and we just proved it, into the container, so now I can do all my edits locally, but I can actually run it in whatever container I want. Now, in this particular demo I'm using Node.js, but this applies to PHP, ASP.NET, Python, whatever it is you want to run. So that's an example of the actual commands that we learned about going back to up here that would allow us to link our working directory, the current folder that is on our local machine on a Mac or Windows or even Linux, and we can now link that into the volume that we defined this var/www up in the container. Pretty cool stuff.

Hooking a Volume to ASP.NET Source Code

Now let's take a look at how we can work with volumes in an ASP.NET Core container. So I've already created an MVC project using the dotnet new command. And I'll show you that real quick in case you're new to it. But this is just and out-of-the-box project. It's an MVC project, and what we're going to do is get this running inside of a container. So the first thing I'm going to do is run off to the command line. So let's come on back to the command with Open in Terminal. And then inside of here, what I'd like to do first off is get a container going. Now before I show you that, if you're new to .NET Core at all, you'd have to go to dot.net, and then you can download the SDK it's called, and I have that installed on this machine. And then I could do this, dotnet new, and then we can give it the project type. I did MVC. That actually generated the project that you see right here. What we're going to do, though, is we're going to get this code running, but not locally. We're going to get it running in a container running on my machine. And it's going to be a Linux container. So to do that, the first thing we need is the actual image that we want to work with. And in this case, Microsoft publishes an image that we can actually use to work with this type of thing. So I'm going to go ahead and pull this image onto my machine. I actually already have it, but I'll show you the

command. So we could say docker pull, and instead of going to Docker Hub, we're going to go to the Microsoft Container Registry, mcr.microsoft.com. So that's the domain of the registry. Now we're going to give it a path to what we want from an image standpoint, and we want the dotnet/core/sdk image. And if I hit Enter, this will now pull it down. Now the first time it pulls, you'll get the different layers. It will probably take you a minute or two to get those depending on your bandwidth. And we've already seen how it pulls layers and kind of shows you that output. So now that we have the image, the next thing I want to do is I want to run this, but I want to link it through a volume back to this source code here. So what we're going to do is kind of an interesting take on running the container, link it back to this source code on the left here that's on my local machine. But then I'm going to interact with the running container right through this command prompt here. And the way we're going to do that is with a -it command. This is an interactive, and it stands for TTY, kind of an older term, but, in essence, it's going to allow my terminal window, my terminal shell, to link into the container. And I'll show you that in just a moment. Now the next thing I'm going to do is we're going to say the volume that we want is to our current working directory, print working directory (pwd). Now the next thing I'm going to do is put the name of the folder. I'm going to call it /app. But I could call this anything. It could be foo, foe, fum, whatever your folder is. But we'll just do app in this case. And then I'm going to make that the working directory, the startup directory, so that when the container starts up, it jumps right to the app folder. Then we're going to say the name of the image, and we, again, do dotnet/core/sdk. But then I'm going to say, Hey, I'm going to shell into a bash shell in this container. Now one thing I want to point out really quick here is what port should we run on? Okay, we haven't defined that yet. And also that pwd syntax. So let's briefly talk about that really quickly before we move on. So I kind of purposely left out the port because we do need that. We can put it anywhere in here. I'll put it right after this, let's say. And what do we want to run this on? Well, let's say externally that we're good with 8080, but what's the internal port? Well, it depends on your project actually. We have a launchSettings.json file that they included by default, and this will become the two ports that we look for. Now we're not going to have a Development certificate, though, available. We could do that but not in this particular demo. So I'm going to take out the HTTPS one, and then I don't necessarily just want localhost to work. I want any IP address, so we're going to put a + right there. Now there are many ways you can override the default port where your code runs in the container or even on your local machine. You could update this launchSettings, you could set environment variables, or you could even go to the Program.cs, and there's some syntax you could use there. But this is kind of the easier way based on what they give us. Okay, so now that we know that, we can do 5000 right there. Okay, so that looks good. Now what about this syntax right here though? Well, this is only going to work on Mac or Linux. Okay, so let me show you just real quick. If you had PowerShell, then it would look like this. And that'd be kind of PowerShell. If you are on, let's say, just regular DOS, then you could do this, %cd. So the syntax that you put right here really depends on what command shell or command window you're using. Because I'm on Mac in this demo, I'm going to put the Mac or Linux-type syntax. They're both the same in this case. But be aware that you need to kind of look into that, and you can see from the link down below here, there's a blog post you can go to to learn more about that different syntax across the different operating system command line prompts. Okay, so now we've done that, we have docker run in interactive mode. We know the external and internal ports. We're going to have the volume link from a folder in the container called app back to the code that is in our working directory. That's our DEMOS. You'll see right up top here. The startup directory's app, that's our working directory. We have our image, and then we're going

to fire up an interactive, kind of bash terminal modes. So let's hit Enter. And there we go. So now we are in the container actually, and I can actually do things like ls, and you see how it linked back to my local source code. This code was never in Microsoft's image. How would they know to obviously put this code? They wouldn't. So, instead, that volume linked the app folder, which is our working directory, back to this local source code. So just think of it as you have a container kind of like in a bubble. Somebody poked a hole in the bubble and put a hose in there, and that hose kind of flows back to the directory on your machine. Now we can kind of talk between the two. So now I can do dotnet run, dotnet build. In fact, let's just do a build real quick. Dotnet restore, all those type of commands. There we do. I could do dotnet run, or I can even do dotnet watch run. Let's do a watch run if you're not familiar with that. That will make it so if any files change over here, it will automatically restart the server that's running in the container. Super cool! So let's hit Enter. All right, we'll let this fire up, and then we'll run off to the browser. So notice it's listening on any IP 5000. Okay, and then we just saw we did 8080 to 5000, so we're good there. So let's go ahead and go to the browser. Okay, so let me go from localhost to localhost 8080, and notice it works. This is now running the server, though, in the container even though our source code is local. Now here's what's kind of cool about that. Let me go back to VS Code. Let's go to our index, our home. Instead of just Welcome, let's say Welcome to Docker Volumes or something like that. And let's save. Now watch down below. See how File changed, Index.cshtml. Now it's doing the rebuild, and it restarted the Kestrel server in the container. Pretty cool. So let's go back to the browser. And there we go, Welcome to Docker Volumes. Now, are you going to develop this way? Probably not because, normally, if you already have the .NET tools local, you'll just run dotnet run local. But what this shows is how you literally could link this container or any container to local code. And if you didn't want to install something for whatever reason, you literally wouldn't have to. You could run the container that has your framework, your server, whatever it is, and just have it link back just like we did here. Now coming on back, let's clear. We'll exit. Let's do docker ps -a, and there we go. We have an exited container, you'll notice, f5b, so let's do docker rm f5b. Now I could do this to remove volumes, but because we created the volume, it wasn't a Docker kind of allocated volume. We did it. We said pwd, for example. Then this -v won't have any effect because we created the volume, not Docker. I'll talk about this a little bit later as well. But in this case, I'll just do f5b. There we go. Let's run docker ps -a again, and it's gone. And so imagine that I didn't actually create this code. I checked it out from somewhere, but I don't have the SDK installed on my machine. I could just get this image going and literally do live development against it without ever installing it. Now, again, that's not what I'd do in the real world because I already have .NET and the tools installed on my machine. But there are a lot of powerful techniques here that you can use in various scenarios. Keep in mind, volumes could also be just to store log files the server writes out. There are all kinds of reasons you might use volumes.

Removing Containers and Volumes

In some of the earlier demos, I showed how you can clean up volumes as you delete your containers. So I want to reiterate and go through some examples of when you need to do that and when you really don't need to do that. So if you run a container, and as you do that you actually add a volume to that container, like we saw earlier, and you only specify one part to the volume, as you see here, I just have /var/www, then in this case, Docker is actually going to manage the volume location where it reads and writes. And so we're not specifying where

our source code is or where to write. We're going to let Docker figure that out. All we're doing is saying that the container has a volume of var/www, and then Docker is going to do the magic that actually creates that folder and mounts it on the host machine. Now, in those cases, which definitely will be reality in some production or staging scenarios where maybe you write log files or things like that, then when that container goes away, if this Node container, for instance, needs to stop and then be deleted, we'll probably want to clean up that volume, because otherwise, you kind of have some dangling files, and it eats up some hard drive space. And I mentioned this in some of the earlier demos, but just want to reiterate. So, if you run docker inspect on your container, I showed earlier that you can actually see the mounting location, and you'll see the Source property in the Mounts property. So you'll notice a nested object with Source and Destination. Now, if you see that it's mounting it and that Docker is taking care of it, and that means again you did -v with just one piece, not two pieces, as you define the volume, then when you're down to your last container, you're going to want to remove this so you don't waste any hard drive space. So, as I showed earlier in some of the demonstrations, you can just simply say docker rm -v, and that will say, in addition to removing the container using the volume, let's also remove the Docker managed volume. Now, as I showed in one of the demos, if you do this and your volume has two parts, you have the container volume name, but it actually points to a folder you specify, like your source code, doing - v is not going to delete like your source code. It's going to leave it all there. So this is really only needed when you specify a volume and you let Docker manage the location on the host machine of where that volume lives. Now, if any other containers are using the volume, you'd only want to run this when you're down to your last container using it. And then it would go ahead and clean that up because obviously some other containers, if they need it, you don't want to get rid of it. So that's a quick review on volumes and the need to clean those up in cases where you delete a container and where you defined a volume that Docker actually manages on the host machine.

Summary

In this module, you've taken a look at the layered file system, and you've seen how images are composed of layers and how containers are really the same thing, but they have a thin read/write layer that sits on top of all the other layers provided by the image. Now we talked about that because there may be times when you want to read or write specifically to store some information, but you want it to stay around. And so we learned about how we can hook source code and how we can even write if we wanted to do like log files or database files into things called volumes. And we learned that volumes are persisted even if a container is deleted. So to do that, we can use the docker run command and specify -v, and then we can either do a Docker managed volume, or we can specify a folder where the volume and the container points to that might have things like our source code. Now, as mentioned, volumes are persisted on the Docker host, and that's a good thing because we might have some log files that a container writes out, and even if the container goes away, we don't want those deleted. Or we might have maybe database files or something along those lines. But if you do get down to the last container and you don't need the volume anymore, then we can remove that using the docker remove command, and we can simply say -v and then the container ID or the container alias. So that's an example of how we can actually get our source code linked into a container. And that's a really, really useful feature to know about because now I can get Node or PHP or ASP.NET or whatever it may be up and running as a

container. Don't even have to install anything on my machine. I just have to get that image in that container running, and then I can simply create a volume that links into my source code, and I'm off and running. When I'm done, I could just delete that container, and there's really no trace of it, especially after I get rid of maybe the image. So I hope that clarifies what the layered file system is and how we can use Docker volumes.

Building Custom Images with Dockerfile

Introduction

Up to this point in the course, you've worked a lot with images and containers, but they've been images that were hosted up on Docker Hub, and we've pulled those down. In this module, we're going to focus on building custom images, and you're going to learn about a special text file called a Dockerfile and learn about some of the instructions that you can put in that. Let's jump into the full agenda. So we're going to start off by talking about what a Dockerfile is, and I'll introduce some of the key instructions you're going to need to know about and explain the general process of how it works for building custom images. From there, we're actually going to create several types of custom Dockerfiles. We'll see the different instructions, and we'll do that on Windows and on Mac. Then we're going to learn some Docker client commands we can run to build a custom image and tag it. And then finally, once we're all done with multiple images, we'll talk about how we can publish an image up to Docker Hub to make it available for us on any other machine, for other team members, or even for the general public if you want. Now the main question that we're trying to address in this module and some others is how do you get source code into a container? And one way we've already learned about. You can create a volume, and you can have a volume that points to your source code on your local development machine. And that's great when you're working in development mode. But in this module, we're going to see how we can actually get our source code into a custom image. That way, that image could be used by other team members or anyone out there in the public if we wanted to set it up that way. So let's go ahead and get started by talking about what is a Dockerfile and what are some of the instructions you need to know to create a custom Dockerfile.

Getting Started with Dockerfile

Developers are quite used to writing instructions in a code file, running those through a compiler, and then outputting some type of binary or other file. Well in the Docker world, we have a very similar type of process that we can follow to create an image and then a running container, and that's to create something called a Dockerfile. Now, a Dockerfile is nothing more than a text file that has instructions in it. Now these instructions, of course, are unique to Docker, and they're defined up in the Docker documentation, but it's a very, very similar process to, if you're writing Java or C# or another compiled language, you'll write some instructions in a file, and then in the developer world we'd run those through a compiler. Well in the Docker world, we'll run them through the Docker client and it has a build command we can run, and then that build command can read through those instructions, generate a layered file system, as we've talked about earlier in the course, and then we have a Docker

image that comes out of this that we can then use, and we can make a container from that. Dockerfile itself, as mentioned, is really just a text file. There's really nothing fancy about it, in fact, it's normally called Dockerfile and oftentimes doesn't even have an extension, but you can name it whatever you want. It's just a text file that we want to feed into the Docker build process. And so it contains some build instructions, which we're going to be looking at, and these build instructions will do things like work with environment variables or copy source code into the image, and more. Now, the instructions that we're going to be doing oftentimes create intermediate images, and these images are kind of behind the scenes images that are cached, and that way if you maybe change an instruction, need to rebuild the image, it won't have to do everything from scratch. Now, there are ways you can override that and not cache anything, but then it'll make your build process take a little bit longer. And, as mentioned, we're going to be using a docker build command to actually convert the text file into an image. Now here's some of the key Dockerfile instructions, and this certainly is not all of them, it's just a few, and I'm going to talk through the high-level look at what do these do. So normally what you'll do is you'll start off by saying I would like to create an image from another image. Now you can create it from scratch, from kind of nothing, but normally you'll create one based on, for instance, a Node.js image or a MongoDB image or PostgreSQL, or something like that. You'll use that as your baseline, and then you'll build on top of that using this layered file system. There's also a way you can define who maintains it, that's a very simple instruction, but you could say your name, and then there's a run command. Now the run command's really important because you can actually have different things defined that are going to be run, and these would be, I want to go out to the internet and grab something, I want to run npm install, dnu restore, those types of things could be actually run using this run instruction. Another really important one is copy. When you're ready to go to production, we learned about earlier in the course you can use volumes for source code, but when you go to production we want to copy that source code into the container oftentimes. There's multiple ways to do it, but that's pretty common, and so we can use the copy instruction to do that. You can also set what is the main entry point for this container. In other words, when you have an exe, or something like that on a system, you can normally double-click it and it has a main entry point that kicks everything off. Very similar here, what is the initial entry point that kicks off the container, for instance? You can also define what the working directory is. This sets the context for where that container is going to run as, for instance, the entry point is run. So I could say what folder has my package.json, and I can do an npm run. You can also expose a port, and this will be the default port that the container would then run internally with, define environment variables, these environment variables can be used then in that container, and then we can even define volumes. And we've already looked at volumes in the course, but you can now define the actual volume and control how it stores that on the host system for that container, as we've already talked about with volumes. Now let's take a look at what a Dockerfile actually looks like then. That's a few of the commands, but it kind of helps to see them in action. So, first off you could say from, and this will always be the very first instruction that you're going to put at the top of your Dockerfile, and I'm going to say FROM node build an image. And this will grab Node as the base file system and then add additional layers on top of that. Now I could say the maintainer, in this case I obviously put me, but this is where you could put yourself, you could put your email, things like that. Then we can say I'd like to copy my source code from my current folder I'm building from, in this case dot, and I'm telling Docker in this case that when you build the image, copy that source code into the var/www folder, which is just one I of course made up. What that will do now is this layered file system will have a layer in it

that's going to be just for our source code, and that will be the copy command or instruction. We can then set the var/www as our work directory because we might want to run some different commands like npm install if it was Node, we can define the port we'd like to expose that the container actually runs with, and we can also define an entry point. In this case I'm saying that the node command and server.js is my initial entry point into this, but of course that could be whatever you want for your chosen technology. So that's a quick look at some of the different instructions, and as mentioned, there are more, but we're focusing on the ones that you really need to know to get started with as a developer. But what we're going to do now is take a look at building custom Dockerfiles from scratch with some different technologies.

Creating a Custom Node.js Dockerfile

Let's assume that you just got back from a team meeting, and you've now been tasked with making a custom Docker image that the team could use, and specifically, you need to build a Node.js image. Now to do that, we're going to need to build a custom Dockerfile, and we're going to need to add instructions into it. Now, once we're done with that, I'll show you a little bit later, we can then use Docker client to actually build that into an image. And then, of course, we can convert the image into a running container. So let's take a look at how we can do this. So I have some code here for a Node.js Express site, and this is the same one generated from Express generator, and I want to call out one thing in the package.json file. You'll notice that we have an npm start command that can be run, and when that runs, it actually runs the node command and then points to a file called www here. Alright, that's going to come into play in just a moment as we make this Dockerfile. Now, the next thing I have in here is I've already added an empty file called Dockerfile. Now it turns out you can actually name it whatever you want. This is the standard format, but if I wanted to rename it to something like node.dockerfile just to give it an extension, I could definitely do that. And when I have just one Dockerfile in a project, then I'll usually just go with the de facto standard, which is this Dockerfile, this one here. But if you have multiple or you just want to give it a more explicit name, then you certainly can rename it. It's just a text file. Now, the first thing we're going to do is use the from instruction. Now the from instruction instructs Docker, I want to build this particular image that we're going to make from another base image, and because we're going to be doing Node.js in this example, I'm going to base it on the official node repository image that's up on hub.docker.com. Now this particular image has a lot of different versions. So you could do this alone, and that would be like doing this, latest, and that will always grab the latest version of Node every time you rebuild the image. Now that could be good, that could be bad because, you know, it could be you don't want to move forward with the latest version, but you could also come in and we could specify a different version, for instance, if we wanted as well. Now I'm going to go ahead and go with the latest one here, and I do like to put latest in cases where I want to grab the latest, because it makes it really obvious, even though, as I mentioned, this is the default that will grab the latest, but I like to be explicit. So we're going to go ahead and do that. Now, the next thing I'm going to do is I'm going to say that I'm the maintainer of this particular Dockerfile, and then you can give your name, you could put your email address, whatever you want on this line. So this is a little bit more of just metadata, but it's good to have as other people look at your Dockerfile, maybe they want to get in touch with you for whatever reason. Now the next instruction I'm going to put is called expose. I'm going to say that we

would like this particular image and the container that comes out of this to actually run on 3000, and that's because that's what the Express side by default will run on. Now, when we do Docker run, as you've seen, we can actually map different ports if we want, but this will give the default. And then finally, I'm going to put something called ENTRYPOINT, and the ENTRYPOINT command is when the container actually gets started up, what is the entry point to fire up that container, and for us, it's going to be the npm start command. So I'm going to put npm start. Now something interesting about this, you'll notice I'm putting it in a JSON array. In fact, I have to put the double quotes in this case, because it is a JSON array; it's treating it that way under the covers. Now, I could do this as well, but the normal recommended way that most people will tell you anyway, is to go with something like this. Alright, so there we have it. We now have our very first node image. Now, it hasn't really done a whole lot, because I could have just done a Docker run on the node image up in Docker hub itself. And the only thing I've gained here is I put who the maintainer was, really, not a whole lot. You know, I did put the default entry point. Okay, that could be useful, but there's no source code that's going to be built in this image, so we'd have to use volume support to make that happen. So let's take this up just a notch and see what we can do here with it. So let's say that part of the requirements for making this image was that we needed to copy some source code into it so that when other people on the team run the container, they don't really have to do anything. Maybe it's going to be a Node.js RESTful API, maybe it's just a web app that's just going to be running that will be hit from some other container potentially. So what I'd like to do first is come on in and use another instruction for Dockerfile called COPY, and COPY does kind of what it says. It allows you to copy in whatever you want. It could be an individual file. It could be an entire folder structure, but we're going to copy the entire project that you see over here on the left. That's everything in here. And I'd like to copy that into the var/www, and that's just a folder structure I'm going to go with to say that's what we want our code to run. And what that will do is now bake the source code as a file layer into that layered file system that Docker builds up for images. And so now our code is going to be in there, it's baked in, it's going to be ready to go. We could also then come in and set the work directory. What the work directory allows us to do is set the context for different commands we might want to run. Where does it run them from? Does it run them from the root? Does it run them in this folder called var/www? It's kind of up to you. So I'm going to say, yes, we want to run it from this var/www. Now, the reason that's important is when I use instructions like run, which is another one that's built into Docker and the Dockerfile, then it needs to know, in some cases, the folder where the context is where that should be run. So, for instance, if I run something like npm install, then we probably are going to want to run that where the package.json is located. Had I not put the work directory, then we would have to actually tell it the context of where to run this command so it can find that package.json, and find all these dependencies that it might want to add to get those going. Alright, so the work directory you'll find is actually pretty common and very useful, especially for us as developers, because our images might need to run some specific commands in that folder or that location. Now, the other thing we can do that's related to this work directory is we could say that maybe we want it for whatever reason on the host system where the container is ultimately going to run, and we know how to do that, and that's to use volumes. So I could actually come in and say volume, and then I could give it var/www. Now that alone is going to cause Docker, once the container runs, to mount this particular volume that's in the container, onto a folder or into a folder on the host file system. And we talked about this in a previous module in pretty good depth, but this would set that scenario up, and it's really useful. We might even have multiple volumes. Maybe this

app needs to write to, we'll just pretend there's a logs area. Well, we could set up a volume so that the logs actually stick around, even if the container is deleted, and then maybe brought back up at a later time. So that's what the volume command can do. Now, I'm just going to leave this in here just to show it. Keep in mind that with the Docker run command that we looked at in previous modules in the course, I showed you how we can actually set up a volume; it was the -v switch, and we can point that, for instance, to the source code on our developer machine. But in this case, we're going to go ahead and use the volume just so we can kind of see it here. Alright, now, the last thing we're going to do is let's assume that also that this needs to run with certain environment variables, maybe, for instance, our code. We're going to expose it here, but maybe instead of using this when they run the container, they might specify a different port. Maybe in production, it's port 8080 or something like that. Well, we can use environment variables as well. Now, I normally like to put these up at the top, although they can go in different places. It depends on if you're going to use them in the Dockerfile or not, but in this case, we're just going to make an environment variable, and I'm going to make two of them. We're going to do the node environment, and let's just assume we want production for this particular container, instead of the default, which would be development. Let's go back to production, and then I could right here do another named value pair, as you'll see these environment variables are just the name of the environment variable, and then the value. And I could just do another one, but I'm going to break it into two steps so you can see the separate instructions. Let's say we also want to put a default port, and I'm going to go ahead and leave 3000, that is what Express runs, but this would be an environment variable that your Node.js code can now read from. So now when that container gets fired up, and if this is for production, you could potentially say a different port, maybe for production containers or something like that. Now I'm going to go ahead and match it with the EXPOSE that we have here, but the goal again is just to show you that, yeah, we can do environment variables. So that's an example of a custom Dockerfile that does a few things. Number one, as a review, it pulls in the latest Node.js image. We say who the maintainer is. We define two environment variables that will be in and available to that container. We copy our local source code from here into the image into a folder called var/www, which is also our working directory, and it's going to be set up as a volume, which in this case means the Docker host would actually be where that source code is going to live. But we can override that again with Docker run. We're going to run the npm install command, because we need to get our dependencies installed once that container is going, EXPOSE port 3000 for the internal port for our container, and then we're going to have our entry point as npm start, and that would be an example of a Dockerfile. Now, before we wrap this up, I'm going to clean it up just a little bit, because I don't want to actually put this here and here for the port, because if we are going to be running dynamically based on a port that code loads, and I probably want to expose that same type of thing. So we can actually use environment variables, and I can do something like this. And what the image will do is once this is defined, it will then go and apply that exact value, which in this case will be 3000 right there. And there's a few other spots we could potentially even do that, maybe even for in these areas. But we'll go ahead and leave it right now, because I don't need to set that as an environment variable, but the port might be something. So now the EXPOSE will actually read it from the environment variable value. So that's an example of some of the key Dockerfile instructions you can use. There are certainly many, many others out there, but these are the key ones that you need to get started with. So now what we're going to learn about is how do we take this and actually convert it and build it into an image?

Building a Node.js Image

Once you have your Dockerfile completed and all the instructions are in place, you're going to need to run that through the build process using Docker client, and that's what we're going to look at here. So how do we take that Dockerfile for Node.js and actually make it into an image? Well, it's actually a very simple process. Docker, the client, has a build command you can run, and then what you can do is tag that build, and you'll want to tag each build. Now you can do --tag, or the shortcut that you see here, which is just tag. I prefer the -t, it saves you a couple keystrokes. Now from there, you're going to go in and give your image that you're going to make a tag name. Now, if you go up online and look at all the node images, there's a whole bunch out there, lots of different versions because they're all tagged with the version. Now in this case, I'm just going to say whatever my username is /node, we're just going to say it's good enough for our team, but we could put more details there with version info if we wanted. And then finally, I'm going to give it to build context, which is going to be the folder where it's actually going to run this from that will help find the Dockerfile and do some other things along the way. So let's go ahead and do this with the image that we have. Now, the first thing I'm going to do is I want to get rid of my node_modules here. So I'm going to come in and use a little npm tool I use a lot called rimraf. So the first thing, I'm just going to right-click and open in Terminal, and I'm going to use this rimraf, which is a delete module. And I want to show you that if I get rid of the modules here, that as we build the image, that it will take care of that for us. From here, we can go ahead and build our Dockerfile into an image using the docker build command. Now, I mentioned that you can do -t for the tag, but before we do that, notice that our name, again, is Dockerfile. And that's the default name that the docker build process looks for. But if you do have a different name for the file, it could be dockerfile.dev or node.dockerfile or something like that, then -f will be for the file name, and then I can put the name. Now in this case it's redundant because that's what it looks for anyway, but that's a nice one to know as you might have different file names for your Dockerfiles. Now I can do my tag, we'll give my Docker Hub ID, and we'll give it the name and then the context, the folder in which we're going to run this is dot. So let's go ahead and run it. Now I already have the node image, just as a heads up, already installed locally as a Docker image, so I did that on purpose to speed this process up. So aside from the npm install, which it's doing right now, this should go really, really fast. But once this is done, I'm going to show a couple things here that relate to something called intermediate containers. All right, so we're all done. And if we do docker images, you'll see my danwahlin node, and then there's my node base image that it's based upon. But notice that every single instruction generated what's called an intermediate container. And if I go way up to the top, and let me just slide quickly back up, even the environment instructions each generated their own separate intermediate container. Now, what happens is these containers, they won't show up in your Docker images when you run that, but they will be cached behind the scenes so that the next time I do a build, if this instruction, such as the environment one, doesn't change, then the build process can just say, hey, I've already seen that before. Let me just go load the layered file system layer and just include that in the build. So it's very much like source control. Every time you check in a small little thing in source control, it tracks it incrementally. That's exactly what happens with Docker instructions. Now in the case of our environments, I could have put, because I used the equals, if we go back to here, I could have put the port equals 3000 right next to this one up on top, and I mentioned that earlier, and that would have just done one intermediate container, but because I ran them as a separate instruction, it has to do two lookups, and so

those are very quick lookups, not a big deal, but it's important to know that every instruction leads to an intermediate container being created that's ultimately cached behind the scenes. All right, now that we have that done, let's go ahead and try to do the run process. I'm going to do a docker run, I'm going to run this in something we haven't seen much up to this point, in a daemon mode, that way the output of running the container won't actually show up in the console. It will run behind the scenes, and then I can do other things with the console if I want. So we're going to do the port, we'll do 8080:3000, and then we'll put the name of the image, and let's run off to the location. And this will be 99.100:8080. You see it right there. All right, and it looks good. This is showing that the source code we copied in that we did previously in the course is actually being used in the container. That's an example of how we can work with our Docker files, use some of the different instructions, and then use the Docker client commands like build and run.

Creating a Custom ASP.NET Core Dockerfile

Let's take a look at how we can create a custom ASP.NET Core Dockerfile, some of the different options for doing that and some of the different images that are available that we're going to use as the base of that cake that we kind of build on top of. So the first thing I'm going to do is run off to hub.docker.com, and I've already searched for Microsoft .NET Core, and if I scroll down just a little bit, you'll see .NET Core shows up. Now there's a few options here, but we're going to click on this first one, .NET Core, and you'll notice these are the official images for .NET core 2.1, 3.1, and of course this will just increase over time to different versions, so make sure you always check for the latest version as you do this. Scrolling on down though, you'll notice there's some different featured .NET Core images. We have .NET Core SDK; .NET Core ASP.NET, notice that's for production runtime; and then we have some other ones as well that you could use if you're not doing like ASP.NET, for example. They even have some .NET 5.0 images. So, if we move on down, they'll give you some info about this. They'll talk about how to build, and they'll even have a little example of how to run it, but, what we're going to do is use the SDK image first and I'll show you the basics of that. Then we're going to go ahead and look at the runtime image, and I'll show you the basics of that as well. So let's jump into VS Code here. The project that I have loaded is just a standard ASP.NET Core MVC type project. You'll see a Controllers folder, and Models, and Views; and inside of it I already have two Docker files that we're going to look at, but, I'm going to show you how we can actually generate one of these kind of automagically, if you will. Now let's go to our dev.dockerfile here, and this will look pretty familiar based on what we've covered earlier in the course. We have our from, there's our base image, so, mcr.microsoft.com would be the registry, and then we have the .NET Core SDK, and that's the one we want in this case because you'll see we're going to do a dotnet restore and a dotnet watch run, but notice there's no code in this one. So we'd have to set up a volume and you can see an example of that down here in the comments, and we've already looked at that. But walking through it, we have the base, which is our from; we have our label; three environment variables here, notice that we're going to target Port 5000 on this one; our working directory is going to be /app, that again can be whatever you'd like, and then we're going to do a bash command that would run the dotnet restore, followed by the dotnet watch run. Now a little trick I'm going to show you though is we could come down and build the image like we've already seen, but we could also use a VS Code plugin. So you'll notice on the left here, I have a little whale icon for Docker, and you can notice that I have my images,

any running containers would be listed here. I can connect to registries, so if I had a personal registry or a company registry, and then networks, volumes, and you'll notice all this kind of really cool information down here. Now I have quite a few images you'll notice, and if we move on down, here's an mcr.microsoft.com one, there's the SDK one actually. So how do you get this? Well in VS Code, if you go down to the extensions, which is this icon here, you could just type docker, and then you can install the Docker extension from Microsoft. It's a really, really nice extension, totally worth installing, I use it for a fair amount of things. And so once you have it installed, you now will have this little whale icon on the left that would get to your local images, local containers, and then I could even connect to a registry here, such as ACR, Azure Container Registry; Docker Hub; some other one that's maybe local that stores these; GitLab, and you'll notice a few options. Alright, so that's a really nice plugin and I'm going to show you how we can use that to do some cool stuff here. Now, this first image would be more if we're going to do a volume, but let's say we're not going to do that, we would actually like to build a production version of our image. Well, prod.dockerfile in this case would be for that, and this is what we call a multistage build Docker file. This has multiple steps that we can go through to not only build our code in a container, publish it, but then copy that published code to the final image that we want to generate. So literally from starting the build to actually generating the final image that's going to go up to, say, Docker Hub or some other registry, we can do all that right here with this. So let's walk through this. So in the first part, we're just going to find a base which is based on the aspnet runtime. That's a production version and I'm going to alias it as base. That's something you can do in Docker files just to give an alias. Now we're going to in stage two, come down and go to the SDK image, and we're going to alias that as build. Now all this does is it goes and copies in this local project here, the csproj, does a dotnet restore, copies in everything else, sets the working directory, and then does a build. You'll notice we're going to build a release mode, we're going to output to a folder called app, in this case. So we're literally building the code using containers, ultimately, behind the scenes. Now from there, we're then going to create this from build as publish, okay, and remember, build is our alias up here that's going to have our build code. Now we're going to run the dotnet publish command, and we're going to output that to the app folder. So, what'll end up happening is ultimately the app folder will get our distribution code that we'd want, our DLLs and things like that. Now the final stage of this is we're going to create this from base. Now remember, base is ASP.NET, that's our runtime production image. We're going to generate that, and we're going to put the environment variable for the port we want to listen on, you'll probably be changing that, but that's what I have, and then notice that we have our expose, our working directory, and then here is the magic right here. Notice this from=publish, now remember, publish was the one up here that was based on build, and that actually ran the dotnet publish command and updated the app folder with it. From that publish image, we actually want to copy from the app folder to the local folder. Well what's the local folder? It's the working directory. It's also named /app. Now, I could've come in and done this, it would have been the same thing, but since we set the working directory, that would just be duplication, we don't really need that. Now from here, we take the name of the DLL that would be generated, and that is what will actually be run in this case. Now, if you're looking at that going, gosh, there's no way I could memorize that and do all that by hand. Well, I don't really either, to be honest, and I'm going to show you a really cool trick here. So remember I installed the Docker extension that I talked about that gets me to all these things. Well, it can do more than that. If you hit Shift+Command+P or Shift+Ctrl+P - Shift+Command would be on Mac, Shift+Ctrl would be on Windows - and type docker, you're going to notice all kinds of stuff that shows up. Now, the one I want is

docker add, and notice this Docker Files to Workspace. What this will do is actually generate Docker files based on your target framework. So I'm going to click on that, now it's going to say okay, well, what are you targeting? Well, I'm going to say ASP.NET Core, and now notice I get an error down here. Now, if we expand this error, it says please generate a build task. So we need to do the same thing. We need to get back to the command palette and type this .NET Generate. So let's do that. I'll go back to the command palette, .NET Generate Assets for Build, I'm going to click on that, and there we go. Now what that just did is updated a launch and a task JSON file, which basically can handle the build tasks. So now that's there, let's go back to the command pallet, we'll type docker again, and I'm going to do this Add Docker Files to Workspace. We'll pick ASP.NET Core, and now from here, we can pick either Linux or Windows. Now I'm going to pick Linux. Do you want to include a Docker Compose? I'm going to say no in this case, we'll talk about Docker Compose later, and then what are your ports? Well, for us it's 5000, and then we hit Enter. Now what this just did is generated something very, very similar to what you saw earlier in kind of my custom one. This is a multistage build where we have a base; we have our build image, which does our build; we have our publish and then we have our final down here, and then base this name based on what's in my csproj, you'll see that name matches up right there. Now, what's really cool about this is number one, I didn't have to write it. This is only for production though, so it's important to note that any time you see ASP.NET, that's going to be for production. Now we have pulled in the latest version of the image at this time. That will obviously change over time, but that gives you a nice starting point, again depending on what version you're working with. Now it's pretty much identical to what you're going to see here, it's really close, and then I even have some steps on how you can build this, but with the extension, we could actually come on in like this, and you'll notice right here at the bottom, Build Image, and if I click that, it'll actually try to build that image right now on my machine and I don't even have to really know what I'm doing. Now, it's pulling down some layers that were missing, it's now doing the multistage build, the publish, all that fun stuff. We're going to look at this a little bit more manually in the next section, but, that's how easy it would be to get started with this. Now I'm going to Ctrl+C to stop this, we're not going to really use it anymore, but that would be an example of generating a Docker file that you don't even have to write the code, really all I did again is either Shift+Command+P to get to the command palette or Shift+Ctrl+P, and if the Docker extension's installed, just type docker and you'll see there's all kinds of Docker commands. But, that's an example of a more production Docker file, and then I showed you a more development Docker file, where you might want to create a volume to link back to your code. So now you kind of have both ways of doing it. So from here, let's see how we can take this and build it, and then we'll start looking at this.

Building an ASP.NET Core Image

Now that we have some Docker files available for ASP.NET Core, let's take a look at how we could build those. The normal command you're going to use to convert a Docker file into a image is Docker build. Now what you're going to do, though, is tag it, and the tag is going to contain a couple things normally. Number one, if you're going to publish this up to a registry such as Docker Hub or Azure Container Registry or something like that, you're going to have a username, typically. It could be your team's name, maybe it's you, for instance, on Docker Hub. I use danwahlin, that's my username, and that's going to be the first part. Now that is

technically optional if you're just working locally, but as you start to work with registries out there, you'll want a username. Then we can put a slash, and then we can put the name of the actual image, and then we can even put a version, and I'll show you some of that coming up. So this tag that you see here, this -t, that's shortcut for --tag. Then you have the tag name, and then you have to tell it where is the actual Dockerfile you're going to be building from. You might have to use a -f switch; -f would be the file name, and give it a path, but if it's just called Dockerfile, you could just put the dot that you have right here. But if it's not, then you have to give it, where is the Dockerfile? What's the path to it, if it's named differently? So that's what we can do to actually build an image; it's very, very simple. So let's take a look at that. So coming back in, we have our dev version where we can hook this up to the local source code through a volume, and this has everything I kind of need to run the SDK in a container. We saw something like this earlier. So I could come in and do this. We could copy that. Let's go ahead and open a terminal. Now we could copy that in and run it, but that's going to run the Dockerfile, because we didn't give it another name. Now, before I do all this, let me go ahead and let's just get rid of this for now, and let's do docker build help, --help here. Alright, now if we scroll on up, you're going to notice that we can use a -f. We can give it the name of the Docker file, because the default path is Docker file. So we'll paste that back in, and we'll come on in, and we'll add a -f, and we'll update the image name as well. But we'll give this dev.docker file, or whatever your file name is, and let's put an image name in. And I'm just going to call this aspnetcore-dev. Now, I don't have a username in that case, notice, because I'm not going to publish this. But if I was, like to Docker Hub, it would be danwahlin/ like that, and that way it would be whatever your username is. Now, I can also tag it with a version. We can say this is version 1.0 or 2.0 or whatever it is. If you don't tag it, it becomes what's called latest. Latest is it doesn't really have a version per se, it's just when somebody pulls it, they get whatever the latest version is. That can actually be a little bit dangerous, so I would recommend that you do tag them with versions, but we'll keep it pretty simple here. Alright, so I'll hit Enter and there we go. It just did the build. Let's clear. Let's go to docker images. And there we go. There's our aspnetcore-dev latest, it looks like it's about 705MB, because it's the SDK, so it's going to be a little bit bigger. Now another way we could have done this is instead of first off, looking there for the images, if we come on in here, we could also find it. There's aspnetcore-dev, latest. So aspnetcore-dev, let's remember. Now. I could have done this, though, and I showed you a little bit of this earlier. We could actually right-click and build, and then give it a name, aspnetcore-dev, and then I could give it a version if I wanted, but let's just hit that. Now this will do the same exact thing, but notice if I scroll back up, it actually wrote out the Docker build. It did some other things, but the big one is it did -f dev.dockerfile, and it tagged it, and then it gave it the local folder. So there we go. It just built it. Now we could use what's in this file. So we could come on down, and let's copy this, paste it in, and then we'll get rid of this, and we'll put the name of our image, which is aspnetcore-dev. Now notice we have the app folder linking back volume wise to my local code again. And just as a reminder, what you put there depends on the console you use. So this will work with Linux or Mac or Bash or SH-type shells. But if you're on Windows, it depends. So I showed that a little bit earlier. Alright, so let's hit Enter there, and this should now fire up the container. Now this kind of locked it up, because I didn't do -d for detached, but that's okay. Now if we scroll on up, it's listening on port 5000. We said to go to 8080 to 5000. So let's pull up the browser. Alright, so we'll go to localhost 8080, and there we go. It's working. Now if that didn't work, remember that I talked about that through a properties folder and through the launch settings, you can also control the port that it runs on. So if that's in your project, it may be overriding what was actually included here in the Dockerfile. Last one kind of wins, so

just kind of be careful. If you go to that and it's not working, it's probably because your port is wrong. Because maybe you have a properties with the launch settings, and you need to actually change that port, potentially. Alright, now let's do a docker ps -a, and you'll notice that we have a status of up two minutes ago. It's been up for two minutes. Let's go ahead and stop this one. So we'll do docker stop b8, and docker remove b8. Alright, and then docker ps -a. You can see it's all gone. Alright, now for the production one, to get that one running, I could use the one they generated. Now I'll give you a little gotcha on this. They didn't say the actual port for Kestrel to run on. They did expose port 5000, but if you look in the other prod.dockerfile I have here, notice I have an environment variable. Alright, if you don't put that, then it just is going to default, and the default is 80, and that's fine. You could leave 80 if you'd like, but I just want to point out that again, sometimes you'll bring these up, and you'll just get not found, and you sit there and struggle going, why isn't this working? Probably a port issue. So you'll have to kind of figure out if you're using the right port. Now for this one, I'm going to go ahead and use the Dockerfile they generated, but I could do this one as well. So if I right-click on this, notice I could build it again. So let's go ahead and do that. Now, for this one, it's actually going to name it, kind of based on what the project is. Alright, so notice it's aspnetcoredocker31, because that's what the project name up here was. And that's fine. We'll go ahead and leave that. now if I come into the Docker icon, and let's refresh this. There we go, there's our image, latest. Now I can right-click on that, and I can actually run from here. Now when I do run, notice it's going to start it up, and you'll see that it did this 5000 to 5000, and it looks like it started up. So let's go ahead and do docker ps -a, and it's been up for 11 seconds, but it did 5000, you'll notice. Okay, and then because we exposed 5000, it did 5000 there as well. So let's go back to the browser. So let's come on into :5000 here, and notice it didn't work. Kestrel never ran on 5000; it actually ran on port 80. So let's go back to VS Code and see if we can fix this. Alright, so I'm going to go ahead and stop it. We'll say, docker stop f0. And then if I hit the up arrow here, there's the docker run that it did, but the 5000 on the left is fine, because maybe that's what we want. We did 8080 before, but doesn't really matter, but the problem is this 5000 right here. We never told Kestrel to run on 5000. We don't have a launch settings. We didn't put the environment variable that I showed earlier. So it's kind of defaulting to 80. Let's go ahead and try this again now. Alright, so it's up and running. You can notice right here that it shows us it's running. And from here we can even do things. I can right-click and view logs. Let's go ahead and do that real quick. And notice it's on port 80 inside. And again, this is one of the big challenges you'll run into is sometimes it doesn't work because of the ports. So let's go back to the browser to wrap up. We'll go to 5000, which now forwards to 80, and now it works. So kind of be careful if you use the Docker extension. Sometimes it will assume the port internally is 5000 in this case, but it was actually 80. So you might have to tweak that a little bit in some cases. So that's an example of different ways we can build images. We did a dev image with a volume, and then we did a production image, and I showed how you can do it through the command line or using the Docker extension.

Publishing an Image to Docker Hub

Although you can always build a custom image using the docker build command anywhere you'd like, you may want an easy way to deploy this and pull it down so you don't have to build it every single time. And we can do that by publishing an image up to Docker Hub. And that's what we're going to take a look at here. So the command you'll use is really, really

simple. Number one, you will have to go to hub.docker.com and create an account, and we talked about that earlier actually in the course. Very simple to do, very quick. And then you'll have to run a logging command, and I'm going to be showing that in just a moment. But once you're logged in, it's very, very simple to push your image. All you have to do is say docker push, give it the username and the name of the image, in this case, node, and then that's going to go ahead and push it up to the Docker Registry. So let's take a look at how we can do that with the node image and the ASP.NET Core image that we generated earlier. Now, before we can run a docker push, we will have to log in. So we'll do docker login. Just hit Enter, and then you can put your username and your password. And then it's going to ask for the email you used as well. All right, once it's done that, from here we can go ahead and try to push, and you'll notice it's kind of saved some of our credential information locally. So now I could come in, and let's do docker images again, and we'll do docker push danwahlin/aspnetcore, and then this is actually going to prepare the image and then push it up into Docker Hub. I'll go ahead and let it do that, and we're going to come on over as well to the Mac side, and I have the node image that was created. Now I've already logged in on this machine, so we can again do docker push, my username, and the image tag, and that will go ahead and prepare that. And after a little bit of time after it's done pushing this up, we'll be able to log into the site, and you can actually see your image up there. Now, right now this is going to go ahead and put it into a public repository, and I'll show you that as soon as it's done. It looks like the image is now pushed up to Docker Hub. So we'll go to hub.docker.com, and let me go ahead and log in here. All right, and there we go. So there's my node one that was pushed up, and there's the aspnetcore you can see. So we could click on this, and there won't be much in here because I don't have any descriptions yet, but you could see how somebody could easily pull this and then use that image. And we're going to do that in just a moment. Now, likewise over here on this side, on the Mac side, we've pushed up the Node.js image that was created, and you'll see that's all ready to go. So what I'm going to do is let's go ahead and try to run this now directly from Docker Hub. So we're going to say docker remove image, and we'll give it this f4 image here. All right, so that should be gone. Now we can say docker pull if we'd like, or even docker run if we want to run it. Well, we've already seen that, and now we could put the name of the image. So let's go ahead on this machine and we'll grab the node one, and this is going to pull down the latest version. So you can see some of the layers already exist because they were cached. So you can see how fast that was. Now, I could go in and clear everything out or do kind of a no cache scenario, but you can see that worked. Now, likewise on the same machine, if I wanted to come in and do a docker pull on the aspnetcore one that was also shown in this module, then we can grab that as well. Now this one won't be cached, so it's going to have to pull down everything because I've never run this image on this particular machine. All right, so that will take a moment to run. And then we could, of course, do the same thing over on my Windows box over here as well. It would be the same exact command. We could do docker pull. And since I've already done the aspnetcore one here, let's go ahead and I'll show you how fast this should be. And it should be pretty quick because a lot of the layers are already there. You can see that was extremely quick. Basically what it did is it looked at the IDs for each of those layers and said, hey, I already have these. There's really no need to recreate these because they haven't changed. And that makes it really, really fast as you work with multiple containers and images. So it'll start caching those. So you can see this one's still going; it's going to take a little bit longer. And we're all done. So now I can say docker images, and there we go. We now have the node, the aspnetcore, and then the base node that I had. So that's how easy it is to actually take an image once it's built, push that up to Docker Hub, and now I can pull

that from anywhere. Team members can pull it. Even other people out there, since these are public images right now, could pull it as well. So it's a really, really powerful technology because now it makes it very, very easy to share my exact environment, and we still have a lot more to cover. So that will get us started with custom Docker files, how we can do builds, run containers, and even push images up to Docker Hub. But we're going to start diving into more about linking containers and more as we move along in the course.

Summary

To wrap this module up, we've learned that Dockerfile is nothing more than a simple text file that has specific instructions. You can say what the image that you're going to be creating is based on. That's the FROM instruction. We can run different types of commands, such as npm install or many others. We can define environment variables, set the entry point that will run as you run the container, and much, much more. Now the FROM is where it all starts as mentioned, and this has to go at the very beginning of the file because we have to know what is the base image, and then once we've added the other instructions we can, go ahead and use the Docker build command, tag the image, and then it will be available on your local system. Now, if you do want to make it available either remotely for other team members or maybe even for the public, then we can push that image up to Docker Hub, and that's very easy to do, once you've logged in, using the Docker push command. So now you've seen the process of building custom images, getting those images working as containers, and now, what we're going to talk about moving forward is how we could start to orchestrate multiple containers and have them start communicating with each other.

Communicating between Docker Containers

Introduction

We've learned about how to work with images and get containers up and running, as well as different Docker toolbox tools and how you can use those, but we haven't addressed a really, really important question that you'll certainly encounter as you work with Docker, and that is, how do you communicate between Docker containers? And so that's what we're going to talk about in this module, we're going to focus specifically on how can we do things like have a container that has a web server, talk to a container that maybe has a database or something else along those lines. So we'll start off by talking about the general concept of container linking or container communication, whatever you'd like to call it, and I'll talk about two options that are available that we can use there. We're then going to dive into the first of those two options, which is called legacy linking, and this is a way that we can name our containers and then easily link one container to another container based upon the naming. I'll then show some examples of linking up different containers, and specifically I'm going to show Node.js with MongoDB, then I'm also going to show ASP.NET Core with PostgreSQL. So we'll have some real examples to walk through of how we can get this container communication going. Now Docker also provides a really powerful way to communicate between containers that's related to setting up networks, and so we're going to learn about something called a bridge network or you might hear a container network, and

we'll talk about what that means, the benefits it offers and how you can set it up, and you'll see it's actually really easy to get set up, it doesn't take a lot of time to get going. And then we'll go ahead and show the same examples of Node and Mongo, and ASP.NET Core and PostgreSQL using container networks. And then finally, I'm going to wrap up by talking through the scenario of what if you don't just have two containers, you have three, four, five or more containers that all need to communicate, and we're going to talk about some future parts of the course, and some other techniques we'll be able to do that'll simplify that entire process. So let's go ahead and dive right in and introduce the concept of container linking and communicating between containers.

Getting Started with Container Linking

As you use more and more Docker images and containers, you will certainly run into the need to link them up. We need a web server, for instance, to communicate with the database server or something like that. So, for example, we might have a web server that not only hits a database server but also needs to hit a caching server and maybe even some others potentially. Well, normally, each container will hold its own individual functionality. In other words, you can have a web server container, a database container, a caching type of server container, and maybe others as well. So we need a way for containers to talk to each other because, up to this point, we've only worked with single containers, not with multiple containers kind of orchestrating things together. Now Docker provides two different linking technologies that can be used. The first is now referred to as legacy linking, and you're going to see that this is just done using container names. Under the covers, it creates what's called a bridge network, and within that network, you can communicate between the containers based on the name of each of the containers, and I'll show you how all this works. Now this particular option is still very useful. It's still very easy, actually, to do you'll see. And in a development environment, it's especially easy to set up. But there is another option, especially as you move multiple containers into staging and production areas, and this provides even more power. This second option involves adding containers into a custom bridge network. Now this is a newer option compared to the legacy one anyway. And what this entails is creating a custom bridge network, and this is a type of isolated network, and only containers in that network can communicate with each other. Now this is nice because now you would have a way to create one network for a certain set of containers to communicate, another network for some other containers that they need to communicate, and this allows you to divide things up a little more elegantly than what you can do with the older legacy linking. So throughout this module, I'm going to walk you through, first, the legacy linking and how that works. I'll show you some examples of getting some actual containers communicating. Then we'll move on to using those same exact containers, but we're going to move on to the bridge networking. And I'll show you how you can create a custom bridge network very, very simple, it sounds a lot harder than it really is, and how we can then communicate amongst containers in that bridge network. So let's go ahead and jump right in and talk, first, about legacy linking and how we can name containers so that they can communicate with each other.

Linking Containers by Name

One technique you can use to link up containers to each other is called legacy linking now. And this is a very simple technique where you can give a container a name, and then another container can link to it using that same name. Let's jump into a step-by-step walkthrough of how this works. So the steps to link containers is really basic actually, just a few little command-line switches you'll need to know about, and we'll go through each of these. First off, we're going to need to run a given container that we want to link to with a name. I'll show you how to do that, it's just one little command-line switch you have to add. Now, we can use that name then as we run another container to link those containers together. We're going to take a look at that as well. And then, of course, if you have additional containers, you just kind of repeat and keep going. So you'll add a name, and then link it to the next container, add a name, link it to the next container. So it's like a step one here. All right, so when we do docker run, we've seen that a few times throughout the course, we can do the daemon, that's the -d, that'll make it so it runs in the background, but we can also do --name and then give that particular running container a name. Now, up to this point in the course, we've mainly relied on the ID for the container or the alias that was automatically generated by Docker, but you can give each of your running containers your own custom name. So in this case, we're going to define a name for the container called my-postgres, and that would take care of the basics of naming it. Now, if that's all we did, it's not going to accomplish too much, it's just going to add a name that we can then use to, for instance, remove or stop the container, but now that we've named it, we can go to step two, and we can link up another container to this database container. So, for instance, let's say that we would like to run an aspnetcore container, then we can run it as you see here with the daemon mode, give it a port, pretty standard stuff that we've seen, but we can also come in and link to another container. And we do that with this --link command-line switch. Now, this is the actual name that you saw previously, my-postgres, and then we can even give it an alias that we use internally in the aspnetcore container that's going to be running. So, as we connect, we can use this postgres alias in our database connection string, for example. So that's really all you have to do to link a container to another container that's already running. Now, step three would be we just keep going if you have more and more containers, so you'd start another container, give it a name, link it to the next container, and then repeat. So normally the containers that you're going to link to, they'll typically be started up first with the docker run command, and then once you're done with that, you can then use the --link command-line switch to link any other containers by name to those containers. So now that we've seen how we can do this with Docker client on the command line, let's go ahead and take a look at linking up different types of containers across different technologies.

Linking Node.js and MongoDB Containers

In this section, we're going to take a look at how we can link a Node.js container to a MongoDB container and the Docker technology that makes this linking possible. So I've already loaded up a Node.js project that hits MongoDB. I'm just going to walk you through the fundamentals of what happens here to show you that we are, indeed, going to be inserting data into Mongo and then pulling that data back out. So first off, I have a config folder, and this just stores the connection string type of info, so I have the host and the database. And you'll notice this name, mongodb. Now, I didn't pull that out of thin air. That's actually what

we're going to be naming the MongoDB container. So we'll come back to that in just a bit. Now, we're also going to be calling a dbSeeder, and that calls up into this dataSeeder, and you'll notice that we have some Docker commands here. Now, this is just a custom object I made in Node.js. It just has some custom properties that I'm going to insert. Could've picked anything. But I'm going to insert a Docker command, a description, and then some examples of using that command so we could kind of pretend this is like a help database or something like that. And then I'm going to save it here. And then I'll also create a Docker command, this time ps, and we'll run some examples of that. So it's just some basic sample data that we're going to insert into Mongo using the Node application that you see here. Alright, so that's kind of the fundamentals of the app itself, and it'll just write out those commands to the home page. Now, the next thing I'm going to show you is this node.dockerfile. The actual set of instructions that you see here shouldn't surprise you. We're going to copy the source code into a folder on the container, set that as the working directory, run npm install, and then start up the server. But you'll notice at the top here I have some instructions on how to link everything up, because we want to link again Node.js as a separate container to Mongo, which is its own container. So the first thing we're going to do is we need to convert this into an image, and so let's go ahead and do that. I'm just going to copy this down. And we'll pull this up and just paste that right in there and build it. Now, this should be cached, so it should be pretty fast to do. All right, so we're all done there. And if we do docker images, you'll notice that I have my custom image, I already have a node image, and there's mongo. So we're ready to go there. Now, the next thing we're going to do is we're going to run the Mongo image, but you'll notice that in the run, I'm running it first off in daemon mode, so in a background mode, but I'm also giving it a name, and we really haven't done that much up to this point. So let's see what that does. So I want to paste this down, and we'll get this mongo going. All right, so let's run docker ps, and you'll notice that it's up and running, but you'll notice the name here is now the name that I chose, as you'll see right up here. So the my-mongodb, It could be useful in this case, if you just want to start and stop the container and don't really want to use the ID that we have, but it's also very useful as we want to link containers. And that's where we're really going to use the name here. So the next thing we're going to do, then, is we need to start up node as a container, but we want to link it into this my-mongodb. So let's go ahead and paste this command in, and before we run it, let's talk about it real quick. So we'll do the standard docker run in daemon mode, external port of 3000, internal port of 3000 for the container. But here's the magic. We're going to link to my-mongodb, which, of course, is the name that we gave Mongo that you can see here, and I'm going to give it an alias, though, in the node container of mongodb. Now, remember, when it came to the connection string, if you will, MongoDB was used as the host name, not localhost or an IP in this case, the actual name that was assigned to the container. So that name now is really, really important. Now we didn't have to alias it. We could have just used this external name as well, but we're going to go ahead and go with that here. So let's start that up. We'll run docker ps, and now you can see that we have two containers up and running that are hopefully linked here. So let's run off to the browser, and I already have the IP address for my Virtual Box machine and that port that you just saw, so let's hit it. And it looks like it's running, but, you know, we didn't get any data yet, and that's expected because I didn't run the dbSeeder. So I need to run this dbSeeder now in the node container because that's not something I set up when the server.js fired up. So, I kind of did that on purpose so I could show you another Docker command that's very useful. It's called docker exec. This allows us to execute a command in a running container. I need to know the container, though, so let's do docker ps. Let's just go with d6 here, that would be a little easier, so I'm going to

say docker exec. We want to execute this command in the d6 for the ID, and then I want to run node dbSeeder.js, and I have that set up so that you can run it directly as a module. And that should now insert some data into this MongoDB database, and there is the name of the database. So the server is MongoDB, the database is funWithDocker. All right, so we should have some data in there. Let's run on back and refresh, and this will now hit it, and there we go. So it looks like we now are able to pull that data that was inserted and were able to render it using express in this case. So, that's an example of some of the different commands that you can actually run to, first off, name a container, then reference that name using --link in this case, give it an alias, and then we can use that alias in the linked container, and that makes it really easy now for Node.js to call MongoDB.

Linking ASP.NET Core and PostgreSQL Containers

Earlier, we saw how we can link Node.js to MongoDB. In this section, we're going to talk about how we can link up ASP.NET Core to PostgreSQL. So to get started, I have an ASP.NET Core application. It's an MVC app, and it uses Entity Framework Core. Now, one of the first things it has is a DbSeeder. And so if I come into here, you're going to notice that I seed it with the Docker commands, if I scroll on down that you saw a little bit earlier with the Node.js example. We're just going to use some different containers this time. Now, in addition to that, it has our DB context for this. So you'll notice we have a DockerCommandsDbContext. And this is pretty standard stuff, very basic. It just has a collection of DockerCommands that are going to be seeded into the database. And then if we go into our Startup file, you'll notice that we add Entity Framework support, and this is the extension method right here for calling into PostgreSQL. Now moving on down at the very bottom, you're also going to see at the very bottom of the middleware that here is the dockerCommandsDbSeeder, and then, although I'm not going to go into it much here, this also has a single page application, a spot application available as well, and it shows customers. So there's what we're doing the seeding, and that's the general flow of the app, if you will. Now to get started using it, we can go to aspnetcore.dockerfile. And this is pretty standard based on what we've seen. We have our mcr.microsoft.com/dotnet/core/sdk. We have the label, environment variable for the port we want to use, our working directory. We're going to copy all the code in, this is a real simple example, to the Working Directory folder. Expose that port, and then, in this case, because it's an SDK image, we're going to do a .NET restore and a .NET run. So this wouldn't be used for production, of course, but it's a nice and easy way to quickly build this. Now at the bottom, I put some comments here for running this particular demo. Legacy linking, just to reiterate, is, well, it's Legacy. That's why they call it Legacy linking. And so while it's older, you may come across it, though, and you may even have a scenario at work where you have to use it, potentially, because that might be how it was set up. That's really the reason we're covering this. Although what I'm going to show you here isn't necessarily the preferred way these days, that'll be coming up next when we talk about networks, it is viable, and it does work, and you could do it with ASP.NET Core. So, the first thing I'm going to do is just a standard build. Let's go ahead and do that right here. And I've already run this, so it'll be superfast, you'll see. Now the next thing is I'm going to start up my PostgreSQL image, and we're going to pull that down, which I already have, and then we're going to run it. But notice that I'm giving it a name of my-postgres. Now, it could be anything again. We could just call it postgres, for example, if we wanted. But down below, you're going to see where I link to it. I'm going to alias that. Okay, so postgres is actually what we want to call, but my linking name is

going to be my-postgres because it's based on the name up here. Now I want to emphasize you don't have to do that, but you may come across this, and then you'll know why there's a colon between these two. It's really just mapping kind of an alias, a name to the actual target. Now the other thing we do is we add an environment variable. And in this case, I'm just setting the password of the database. Please don't use password, by the way, for real, but for the demo it's fine. And this will actually pass that to the startup of the container. So when the database first comes up, it will read this and actually use it as the password. Now the rest of this you've already seen, you'll see a docker run in detached mode, port external is 5000, internal is 5000. And then here's our linking, which is really the core of what we're talking about here. And then there's the image I just built. So, first thing I'll do is let's get this running, our PostgreSQL. All right, so that should be started up. Now let's get our aspnetcore, and that started up. Now let me clear this and we'll do a docker ps. All right, so you'll notice both are up. Here's our aspnetcore, here's our postgres. Now, look at the names here. They gave it a name because I didn't name the aspnetcore container, and they gave it angry_franklin. They come up with these really bizarre, random names. But here's our name, my-postgres. Now, what's important here, though, is if we go into our connection string settings. So if I come on in to appsettings here, come on down to our DockerCommandsConnectionString, notice that the server is postgres; not my-postgres, postgres. Now, that's why we did the alias because we named the container my-postgres, but in reality, we want to call the postgres that's actually running behind the scenes here. Now, since both these are linked up and running, let's run off to the browser now. So I'm going to go to localhost 5000. Now, this is going to load a single page app I mentioned earlier that shows customers, but I'll leave this and go to the Docker commands. And there we go. So you could see it worked. It seeded the database with these Docker commands, and then it linked those two containers together so that first off, it knows that the PostgreSQL container needs to start first, then ASP.NET Core. Now as a heads up, it's not going to wait. There's no way for Docker to know when Postgres or Mongo or any of these especially databases are finished loading. So if you ever do have code that has to seed, maybe lookup table data, for example, you might call, and it fails because the database hasn't finished loading in the container yet. All linking does is make sure that they start in the proper order. It doesn't guarantee the database is done, though, so that's something to be aware of, and that means you might have to have some try catch type code and some retries if you're ever seeding something, especially in development. So that's an example of how we could do linking with ASP.NET Core and Postgres. You can see it's very similar to what we did earlier with Node and MongoDB. So from here, let's move on to the more modern way to do this, and that's going to be networks.

Getting Started with Container Networks

You've seen how we can link up containers using the name of a container and how that allows us to communicate between, for instance, a web server and a database server. But Docker does provide a different technique that can be used that also provides additional functionality, and that's what we're going to talk about here. So what we're going to cover is something called container networks or bridge networks. Now to understand this, think of a Docker host. Now, this could be a Linux box up in the cloud, it could be VirtualBox running locally with that Linux box in it, wherever it may be. And then in that Linux box, you have these different containers that need to talk with each other. And so to do that, we could use naming, but anything that knows the name could automatically get to that container by the

name. And while that's a good thing, especially I think in the development environment, it's very easy to get started with and to use, once you start having a whole bunch of containers running, you might want to start to isolate those containers so that you have to be in the same group, if you will. Well, we don't call it a group, but we do call it a network, or a bridge network is the official term you'll see in the Docker documentation. And the way it works is you can, through Docker client, create an isolated network, and you just give it a name. It's a very simple command that I'll show you coming up here in a moment. Any container that's run in that isolated network can communicate with other containers in that same isolated network, and they do so by name. That's why we took a look at the legacy linking type of container naming and linking earlier. That means I could have one set up here, maybe this is a Node.js server talking to MongoDB, whereas I might have a separate, isolated network with Postgres, ASP.NET Core, and some other type of infrastructure set up there for containers. So this is nice because I can actually now group the containers into their own isolated network, and that allows me to isolate them much more in who they're allowed to communicate with as far as their container friends, if you will. The steps to follow to create a container network are actually very straightforward, and the commands you're going to run with Docker client are also very easy. So the first thing we'll do is we need to create a custom bridge network, and we'll give that a name. Now, once you've set up your custom bridge network and given it a name, then you can start the containers up using the standard docker run, but we can specify what isolated network to run in. Now it is possible for a container to run in more than one network, and that would allow it to communicate with multiple containers that might be kind of cross group, if you will, cross isolated network. Now we're going to focus just on one isolated network in this particular example and the examples that follow, but you can definitely do some more advanced things if you'd like there. So let's walk through the steps here real quick. So step one involves creating a custom bridge network. And the way we do that is we use the Docker client, and we use the network command. And we could say, hey Docker, I'd like to create a new network. I'd like to use the bridge as the driver, and there's a bunch of different drivers you can do as mentioned, even cross host is possible, and more. And then I'm going to name the custom network. Now, I gave it a real basic name of isolated_network, but it could literally be whatever you want. This is just like naming an image or naming a container when you run it, you can come up with whatever name you want here. Now, that's it. Now, what that will do out of the box is not a whole lot because it just creates this isolated network, but at this point, nothing's in it. So step two involves then running your containers, but specifying that I'd like to run that container in a specific network, and notice that I'm now saying I'm going to run it in isolated_network, which of course is what we just saw that was created. Now we've said what network we want this container to run in, but how would another container in the same network call into this container? And the answer there is we do just like we did earlier with the legacy linking and we give it a name. So every container that you want to link up will have a name. So in this case, I named it just plain old mongodb. Now, the connection string for a web container that's also in the isolated_network could then call into MongoDB by using a server name of mongodb because that's what the container name is. So I won't have to use the --link that we saw earlier with the legacy linking, and you're going to see all this coming up with an example in just a moment, but all I have to do is just give every container that I want to link to a name. As long as they're in the same isolated network in this case, I can now reference that name just like we saw earlier, and then I'm off and running. I can hit a database, a caching server, or whatever it may be. Now it's important to note that the Docker documentation doesn't actually refer to this technique with the bridge and the container

networking as linking. That's a term I like to use because it just makes sense. We want to link one container to another, but in this world, really, we would just call it communicate between one container and another container. Now to wrap this up, I also want to mention that linking, as far as the legacy linking, is actually not supported in this world. We don't need it of course. We have our isolated network and we can just use that directly. So now that you've seen an example of what this bridge network or container networking looks like, let's jump into the samples that we already saw earlier with Node and Mongo and ASP.NET Core and Postgres. Let's see how we can change those up to use this technique.

Container Networks in Action

Let's jump into an example of creating a custom container network using the bridge driver and then adding some containers into that network so they can communicate. So what I'm going to do is the same exact demonstration I showed earlier with the legacy linking, but we're going to do this with our own custom bridge network. Now I've updated the comments here and added two options. So Option 1 is what we looked at earlier, and this is the legacy linking that I showed. But Option 2, which is the new one, is we're going to create our own network. I'm going to call it again isolated_network, but you would normally give it a more specific name, probably based on the containers that are going to be in that network. Before I run this though, let me come back to the command prompt here, and I'll show you another Docker client command, and it's called network, and we can do ls, and we can list the networks. And you'll notice currently that I have none, host, and bridge, and it shows these different drivers. Well, we're going to be creating some containers in a custom bridge network so we can communicate locally on this host. And so to do that, we first need to create the network. So I'm just going to grab this command here, and we'll run this. And it gives an ID, and now I can run the same commander earlier, docker network ls, and there we go. You can see my isolated_network, and it's the bridge driver. Now what's interesting about this is I can inspect the network is well. So I can say docker network inspect, and I can give it the name of isolated_network. And this gives me some information, but I want to point out currently there's no containers in there. So it does have some information about the subnet and the gateway and some other info up here on the ID, but it's really not very useful at this point. All right, so we need to run some containers in that network, and we're going to do that using the --net switch that I showed a little bit earlier. So the first one I'm going to start is the MongoDB container. So we'll paste that in, and that's going to fire that up. Now that's in the network. So we should build a now do a docker network inspect on our network, isolated_network. And now you'll notice in the containers that we have mongodb listed. And only the items that show up in here are going to be available. So this is actually pretty cool to work with. Now we'll come back and we'll start up our Node container. All right, same thing. This will now add it. And when we do our docker ps, we should see those both running. All right, now I can go to the browser, and I didn't load the sample data here, but let's just refresh. And we should see this Docker Commands show up once it loads up here. All right, and there we go. Now I've already shown earlier in a previous demo that if we want, we can do this docker exec, and this will run against the name that you see here of the container. So it made it a little bit easier. You don't have to know the container ID now. Go ahead and run that, and that starts it up, and then I can just stop to get out. Now the MongoDB database should have some data, and there we go. We're now able to run that. So that's an example of how we can use our container networking or bridge networking. It really

depends on how you want to look at it, but the official term is container networking with a bridge driver. And that's how we can have multiple containers communicate with each other in a way that isolates them to this custom network container that we created. Pretty cool stuff. Now that you've seen that, let's do the same thing with the ASP.NET Core and PostgreSQL. So I'm going to run through this want to little more quickly because we've already seen it. But if I run in and say docker network ls, you'll notice I have kind of the standard items here, and now I'm on the Mac side versus the last one was on the Windows side. So we can again create our custom network. We'll paste that in. There we go. So now we can run our docker network ls, and there we go. It's in there. But if I ran the inspect, it would be empty, of course, as far as the containers. All right, so from here, we'll go ahead and now we'll start up our database container, and then we'll go ahead and start up our web server container that wants to communicate with that. All right, we're off and running, so let's make sure they're started. All right, both are up it looks like over here. So we can come back over and let me refresh this particular IP and port, and we should see the same type of page on this particular browser. All right, there we go. So there's ASP.NET Core again with Postgres. But again, this time they're running inside of their own network, so let's just prove that one more time by doing docker network inspect. And then the name of the network was isolated_network All right, and you can see we have two containers. There's aspnetcoreapp, and there's our postgres. So the name is actually the name that was used in the connection string up in here. And likewise on the MongoDB side, the name of that container, of course, was used in the connection string. So this is the preferred route moving forward with Docker as you are definitely moving to staging and production. Now I'd say in development, I don't know that it matters quite as much because you may not even need a network, but it's just as easy I think to set up a network as it is to link with the legacy linking. So I'll let you kind of debate the merits there. Either one works. But that's an example of how we can do this with container networks.

Linking Multiple Containers

As you've walked through the different samples in this module, you might have wondered, do I really have to type so many commands to link up multiple containers to each other? Obviously, if you only have two or so containers, it's not that big of a deal, but as you start adding more and more and more, it starts to convolute things and definitely make it a little bit more challenging to get those containers up and running and all connected and communicating. So the good news is there is an easier way. If you do have the scenario where you have a web server and a database and a caching server and more, then in the next module, we're going to learn how we can apply all the different topics we've talked about here and put those into something called Docker Compose. And as you could see by their logo for Docker Compose, it's good at juggling and really managing multiple containers in a way that's really, really easy to work with. And so the good news is while you might use some of the commands that I showed throughout this module just to get up one or two containers for sure, if you have requirements that say, hey, we have, you know, four or five containers maybe, or maybe even more, then it is a lot easier to use this other tool that's part of the Docker Toolbox called Docker Compose. And so we're going to be covering that in the next module, something for you to look forward to.

Summary

I hope you have a good idea now about how you can communicate between different containers that you need to get up and running in your develop environment or even in maybe a staging or production environment. So we've learned that Docker containers can communicate in different ways. We can use the legacy linking function, and that's where we use the link command line switch or we can do the networking option as well and that would be one that definitely is very powerful because now you can isolate containers to only be allowed to talk to other very specific containers if you'd like. So the link switch is the one that provides the legacy linking, and of course, the net command line switch is the one that provides the bridge network functionality. Now, I also mentioned that this is all great, but if you start getting past more than two or so of these, then you end up running a lot of commands. then you start trying to come up with ways to batch those to save some time, and the good news is we already have a solution built in the Docker toolbox called Docker Compose, and that's what we'll jump into in the next module.

Managing Containers with Docker Compose

Introduction

We've covered a lot of really fun concepts when it comes to working with Docker in a development environment, but we're now getting to one of my favorite parts of Docker, and that is Docker Compose. Docker Compose provides a great way and a very simple way, you'll see, to get multiple containers up and running with a minimal effort on your part. It's very easy to get started with. The configuration files that we're going to talk about aren't hard to work with, and the commands are even more simple than you've seen up to this point. So let's take a look at the agenda for this module. So we're going to kick things off by talking about what exactly Docker Compose is, and I'll kind of make the case for why we need it, especially in a development environment. We're then going to introduce a file that you're going to need to know about to work with Docker Compose, and it's called docker-compose.yml, or y m l you'll see here. And this is going to be your configuration file that's going to be responsible for taking images and getting them up and running as containers. And you're going to see we're going to call those actually services. Now from there, we're going to talk about some of the commands you can run with a Docker toolbox tool called Docker Compose. So we've seen Docker Machine. We've seen Docker Client, and Docker Compose is yet another tool that you can run a few commands with to do all kinds of great things that are very productive and efficient. Now once we get through the overview of what it is and how the configuration file works and how to run some commands, we'll take some of the images and containers that we worked with earlier in the course, and we'll see how we can very easily get those up and running and even communicating with each other as well. Then, from there, we're going to wrap up the module by walking through a more robust example. Earlier in the module, I'm going to introduce a scenario where we might have a bunch of services. In our case, it's going to be about six services that we need to get up and running for our development environment. I'm going to walk you through the overall development environment services. We'll talk about a custom docker-compose.yml file that can configure these different services, and then we'll talk about how we can manage those

services, and this will include bringing them up, taking them down, removing containers, and some more topics. So let's go ahead and dive right in, and let's take a look at what is Docker Compose and why is it so important, especially in the world of web development environments?

Getting Started with Docker Compose

From a web development standpoint, Docker Compose is definitely one of the more exciting pieces of Docker. It's a great way to automatically manage the lifecycle of your application in the development environment and get it up and running, and stop it, and things like that very, very quickly, and that's what we're going to talk about in this first section. The logo really kind of gives away a lot about what it does. It allows you to have multiple images and then convert those images into containers. Now to do that, though, by hand, which we've pretty much been doing throughout the course up to this point, we've been going into the command line and having to do a manual docker run, and you can see that with a lot of containers that can be a little bit problematic and definitely not very efficient or productive. So the image that you see here from their logo reflects exactly what it does. It allows you to manage multiple containers and the overall lifecycle. Now, if you go look at the official docs, they'll highlight four main areas that it works well. And it's great for the development environment, staging, maybe for production, Docker has some other options you could use there for DevOps like Docker Cloud, but definitely in the development environment it can do these types of things. So as mentioned, it manages the entire application lifecycle, and that includes things like starting, stopping, rebuilding, what they call services. And you're going to see that a service really becomes a running container. So we're still going to be using images behind the scenes that get converted into running containers, but we're going to call those services in the world of Docker Compose, as you'll see as we dig in deeper. It also allows us to view the status of running services, including the log output of all those running services very easily. You don't have to do a command per container to get the logs, you can actually get to all the different container logs at once if you'd like. Now, if you do want to get to one container and do a one-off operation, you want to maybe view the logs for it, or just start and stop that one container or even build it from the image standpoint, then Docker Compose will let you do that as well, so it's a really, really nice way to manage different containers in an app that you're going to be working with. Now, let's talk about the need for Docker Compose. This gives you some high-level kind of 10,000-foot level stuff, but let's dive in a little bit more here. So let's assume that we have a setup in a web app where we have nginx on the frontend, and that's a reverse proxy, we have Redis for caching on the backend, and MongoDB as our data storage, let's assume, and the nginx when a request comes in, let's assume that it also is going to route that into different Node.js servers. Now, again, you could substitute your chosen framework, it could be PHP, ASP.NET, Java, whatever it may be here. Now, as these servers get called, they'll of course call into the database, they'll more than likely then cache some of that data in Redis, and then that's kind of how it proceeds. Now, what's nice, though, is Docker Compose can manage all of these. And you'll see that we have six different containers in this particular case, and you could certainly have a lot more if you have other application servers and things going, and managing those by hand, I don't know that I want to do that. It's a little bit problematic, like I said, not very efficient, not very productive. So Docker Compose has a file that we're going to be talking about called docker-compose, and it's a YAML file. So if you're new to it, don't worry, it's a super, super

simple format. And in this file, you can define all these services and even the relationships between the services. If you remember earlier in the course we talked about linking, and we also talked about networking or bridge networks, and we're going to talk about that as well here as we dive into this docker-compose. So what we're ultimately going to be after here is we're going to make a docker-compose file that can manage the different application services. Now, the services in this case would be the nginx, the Node. the Redis, the Mongo, really they're just containers, of course, at runtime. but in this world of Docker Compose we're going to call and refer to them as services. Now the standard workflow, once you have your Docker files set up, if you have custom Docker files, and your docker-compose.yml file, is you're going to use Docker Compose to then build your services. Now under the covers that's just going to create images like we've been doing all throughout the class. From there, we can then use Docker Compose to start up our services, and then when we're done, we can tear down those services, and stop the containers, and even remove them if you'd like. Throughout the rest of this module, we're going to be talking about these different aspects of the Docker Compose workflow, and we're going to start off, for instance, by talking about the docker-compose.yml file and how you can work with that. Then we'll move into some of the Docker Compose commands that you can run, and then we'll jump into some actual examples of using it and applying it to a development environment.

The docker-compose.yml File

So let's jump into how this YAML file is used and some of the key aspects and instructions that you're going to find in the YAML file. So first off, the docker-compose.yml file defines, as mentioned, all of our services. And so this would be things like, what's the instances of different web servers you might have running, the different frameworks there, Node, PHP, Java, whatever it may be; your database services, caching services, you might have some application server services, and so on, and so forth. And so this will just be a normal text file that on its own is not that useful, but we can run it through a docker-compose build process. And this build process can actually generate images that we can then use to create containers as we run this. Now the docker-compose build process you're going to see is extremely simple, in fact, it's probably the simplest command we've run throughout the entire course. That's why I'm a big fan of Docker Compose. It provides a lot of functionality with just a little bit of work on your part, so we'll be looking at these commands in a moment. Now, that's going to generate, as mentioned, the images. We're going to call these services, though, once they get up and running. And then on a development machine just with one little command I can then build out my services, and then with one other very small command I can then get those services up and running. And so it's very, very nice in the development world, because if I just was given a YAML file with just a few basic commands, I can actually have all my images ready, and then actually convert those into running containers and have these services, if you will, that are actually up and running. Then I can start building my code against those services. So what goes in this docker-compose.yml file? Well, the first thing you'll always see at the top is a version. Now, if you do see a docker-compose file out there, just out on GitHub or out on the web somewhere, and if it doesn't have a version at the very top, then it's probably an old version. The initial versions of docker-compose didn't have a version, but everything moving forward is supposed to have that as the very first thing at the top. Now under the version, you can have different options. You can have things like services, which we'll be talking about, but you can define

other things like volumes and networks as well. Now for our services, this is where we're going to define what is it we want to be running once we build this docker-compose.yml file, and then get all those images up and running as containers. So this is where we define, for instance, Node.js or ASP.NET or Java or PHP. Our databases, our caching servers, and so on, and so forth would go in here. Now, there are a lot of different options for defining these. So, for example, some of the configuration options you can supply include things like the build context. This would be things like what folder do we kind of build from as the context and what Docker file do you want to use to build that particular service, and you'll see this coming up. We can define environment variables, and these environment variables then can be automatically put into that running service, that container, at runtime. So that makes it really nice to swap, for instance, between an app environment of maybe development, to production, and see how your app responds to that. We can also define just an image. Maybe you're not going to build an image, you already have one either local or up in Docker Hub, you just want to use that as the service. We can also associate a given service with a network that's been defined. Now if you'll recall earlier in the course we talked about ways of linking up, if you will, Docker containers at runtime, so for instance linking up a Node.js to MongoDB database. Well, the recommended way to do that, of course, is through networks, and we talked about something called a bridge network and how that can be used to allow these containers to communicate with each other, and we can define those networks and then reference them to link things up in our docker-compose.yml file. We can also expose different ports and define those, and we can even define volumes, including pointing to source code on your local dev machine volumes, and so it makes it really, really easy to hook up a volume into a container at runtime. So let's look at an example that dives a little bit deeper into some of these different options. So, as mentioned, we'll have the version at the top, and then we'll have our services. Now under the services you then name the different services. So I have one here just called node. Now that becomes the name of the service. Now under that, in this case, I say I'm going to have a custom build for a dockerfile called node.dockerfile, and the context is the current folder. So when it builds, use the current folder as kind of starting point, the folder context, of how to reference sub-paths and things. Now I'm also saying that this node service needs to be associated with a -nodeapp network, and this is a bridge network. And that will allow me to put this in a specialized network and then communicate with other services, other containers in that network. Now, here's another service called MongoDB. Now in this case I'm not building from a custom Docker file, I'm going to be using the Mongo image that's up on Docker Hub. So this will cause it to pull it down and then use that image, and then notice I'm adding it to the same network, nodeapp-network. And then you can even define multiple networks. In this case, I define a single network called nodeapp-network, and then it has a driver, which is our bridge type of network. Now, if you're new to the YAML format and you're coming from maybe an XML background or JSON or something like that, then this is definitely very different. You'll notice that there's a little bit of an indentation kind of going on here. And what's nice about this is, number one, you don't have to worry about closing tags, so it's very simple that way. And you also don't have to worry about closing brackets and things, as with JSON. It's just a different way to do it. So you can see it's just a simple file. On its own it's not that useful, but as I teach you and we walk through the different Docker Compose commands, we can take this and convert it into a node service, a mongodb service, and then a network that both of those services are in so they can communicate with each other. So that's a simple example of what could be in a docker-compose.yml file. Now let's look at how we can work with some of the commands that can take this and convert it into images, containers, and services.

Docker Compose Commands

Once you have your docker-compose.yml file available, you can go into the Quickstart Terminal and run the Docker Compose tool and use some different commands that we're going to talk through real quick here. So here are a few of the key commands that we're going to be using in the upcoming sections in this module. First off, we need to build our services into images, and we can do that with the Docker Compose tool, and we can run the build command. That's it, really simple. You'll notice there's not a lot to that, especially if you look back to what we've done when we did builds in the past with just the Docker client. Now once you have your images available, you can then say docker-compose up to start those up as running containers. You can tear them down with the docker-compose down command. And then you can do a lot of other things in addition to that. We can view the logs. We can list the different containers that are running as our services. We can stop all of the different services and then start them back up if we'd like. And then once we've stopped them, we can even remove the different containers that are making up our services. Now we're going to be diving into a lot of these as I move into some of the examples of using them. But let's walk through the fundamentals of the key ones here, the build, the up, and the down. So, earlier, I talked about the Docker workflow involved, building your services, starting them up, and then tearing them down. So let's focus on the build part here. So, as shown earlier, we can come in and say docker-compose build, and that will automatically build or rebuild all of the different service images that we need that are all defined in your docker-compose.yml file. Now this is great because if you had a bunch of services, like I showed earlier, maybe NGINX, Node, Mongo, Redis, and maybe even others, then, with one simple command, you can automatically create all the different images that those services will need to run on your development machine. So it's really, really nice that way. Now you can also build individual services. Oftentimes, as I'm doing this, I make a tweak maybe to a custom Docker file, or maybe there's just a new version of an image that you want up on Docker Hub, and you don't want to rebuild everything. You just want to rebuild one of those services. Well, you can do one-off commands as well, and this would only build or rebuild the Mongo service of course. Now once you have everything built, we can then start those services up. And you saw that's very, very simple to do with our docker-compose up command. That will automatically create the containers and then fire them up, start them up. That includes linking them together if you're doing linking technology or if you're using bridge networks or whatever it may be. So very, very simple. One simple command and you're up and running. Now, again, I want to highlight and compare that to what we've done up to this point in the course where we've had to do individual docker run commands, and some of those commands get a little bit long. This is a lot easier. So this is a great way to simply take a Docker.yml file, do a docker-compose build. Once that's built, we can then say docker-compose up, and we're off and running. Now we can also come in and do a docker-compose up and supply some other command-line arguments here. Maybe there's a particular service we want to bring up individually, such as Node in this case, you'll notice over to the right here. And we don't want any of the other dependencies though. Maybe Node depends on MongoDB or PostgreSQL or something like that. And we don't want to recreate those other services, just the Node one. Well, we can do that with the docker up. That will make sure that the node is brought back up, but we don't re-create the other containers that might be linked into or bridged into the Node container. So we've now looked at building the services, starting up the services, and now let's look at tearing down the services. So the simple command here is docker-compose down, and that automatically will

take all the containers and stop them and then remove them. Now if you don't want to remove them, you could just do docker-compose stop. I showed you that a little bit earlier. But down is really nice in cases where you're kind of done maybe for the day or something like that, and you just don't want those containers hanging around. Maybe you're going to be rebuilding your images anyway. And so you just like to kind of clear all that out. Now if you'd also like to not only stop the containers, remove the containers, but also remove all the images, then you can add some extra switches here. You can do --rmi all would remove all the different images that we have associated with those services. And then you can even remove any volumes associated with those with just a very, very simple command you can see. So, again, you can imagine if you had five containers running or more, this provides a really, really easy way to not only stop those and remove them, but even remove all the different images and all the containers associated with those instead of having to do that individually, like we've done up to this point. Now there are a lot of other commands you can run, but those are the key ones that you're going to start seeing as we look at Docker Compose in action. So let's jump on into the next section here, and let's put this to use.

Docker Compose in Action

Let's take a look at Docker Compose in action, and we're going to work with a custom YAML file, as well as use some of the different Docker Compose commands that we talked about earlier. All right, so I've already opened up a Node.js, MongoDB type of project. This is the same exact one that we saw earlier where we had to manually run some of the different commands to build our images and then run our different containers. So while that works, it's a little bit inefficient, I would argue, and definitely not something I want to have to copy and paste those commands in every time I want to run a container, rebuild an image, or whatever it may be. So I've already created a docker-compose.yml YAML file, and the first thing we're going to do to make it easier to bring up our Node and our Mongo services, which is again, really our containers, is add them and define them into this particular YAML file. So the first thing I'm going to do is we're going to come in and mark the current version that I have to do as of today. Next thing we're going to do is we're going to add the services that we want. And because we're going to have two services here that sort of link up, if you will, we're going to do that through the bridge network. So I'm going to come in and I'm going to name it nodeapp-network. And then we're going to have to say that the driver for this network, since there are different options here, is the bridge one that we've already talked about earlier in the course. All right, so that will take care of having a network that's named nodeapp-network. So that part represents the name. And then the only property I had to put in this case for that was that it was a bridge network. So the next thing we're going to do is come in and define I'm going to call it node, and we're going to do a node service. And I want to build this from the custom Node Dockerfile that I already have. And so I'm going to come in and add a build property, and then it has some subproperties. I'm going to name the first one context. I want to run from the context of where this YAML file is here. So if there's any subfolders I had to get to in the Dockerfile, it would set the context of where that runs from. So that's actually a really important concept. Then the next one is what's the Dockerfile? Well, I'm not using the standard just Dockerfile name. I'm doing node.dockerfile. All right, and that'll take care of that. Now this particular one is going to run on some ports. We're going to do the mapping of the external to the internal. I'm going to do 3000 to 3000. Very similar to what we've done, 3000 on the external, 3000 on the

internal. And then we're also going to need to hook this into our nodeapp-network. So I can say networks and then simply put in--- Every time you see the dash, it's because I could add multiple items here in the YAML file format. We're going to call this nodeapp-network, and that just matches that name right there. All right, so we're kind of off and running with that particular service. So one more time, we're going to call it node. We set the build context to basically the folder where the YAML file is, and then we give it our Dockerfile. We're going to expose the ports that we want to set up and hook it into the network that you see here in this particular case. All right, now the next thing we're going to do is hook in our MongoDB, and this one is not going to be built from a custom image or a custom Dockerfile that I have. It's going to be based on the one that's up in Docker Hub. So I'm just going to list the name of the image there, and let me change that because it's actually just mongo. And then we also need to hook it into the network. So I can just kind of copy and paste this part right here. And were off and running. So that would be an example of creating a custom Docker Compose YAML file. It's not that hard. Really it's just a matter of going to the documentation on docker.com, looking up the Docker Compose YAML file documentation, it's pretty well documented, and then just taking the time to do it. And the nice thing is, once you've done this a few times, you can just start to copy and paste and tweak things between your different YAML files there. All right, so now we have our services defined. We have a node. We have a mongo. They're both in the same nodeapp-network here. And that way, when we run these services and run the up command, then it's going to put both of them in the network so they can talk to each other, and we've seen that earlier again in the course. All right, so let me run off now to the terminal that I already have up for this particular folder and because we have a Docker Compose file, imagine that you just checked this source code, maybe out of your version control, and brought it down to your local machine, and now all I'd have to do to get an environment up and running is say docker-compose build. So I first need to get the images in place. Now this will take just a little bit of time. I have some of this cached, but it should be pretty quick. And you can see it's now done. And then mongo was already there locally if I did docker images. So it didn't have to do anything there. So if we do docker images, you'll notice I have this nodeexpressmongodbdockerapp_node. It kind of named that part. And we could even name it, by the way. That's possible to do too. And in here is my mongo image. And so those are all ready to go. So now the next thing to try to get this going would be to do docker-compose up. Now this is going to go ahead and start both of these up. They just did. And you'll notice though that when I brought it up, it's kind of in log mode right now. And so I'd have to open up a new terminal to really do anything here. Probably not what I want. So let's go ahead and go into here now. I'm going to say docker-compose down, and that's going to go ahead and stop both of those. And it's going to go ahead and remove the containers as well, which is really nice. So we'll let this stop. All right, so that's all stopped now, and you'll notice it also removed, so stopped and removed all in one shot. Now what I didn't like about that is when I did docker-compose up, it kind of blocked us from using the terminal. So let's go ahead and do it, but we want to run in daemon mode. I want to run it behind the scenes. Let's go ahead and try that. All right, now notice they should be up. We'll prove that in a moment. But I can get back to the command prompt. So you've seen that before when we did docker run. So let's go in and do another command, docker-compose, and let's do ps and see what we have running here. And you can see we have two containers. So there's our mongodb. There's our node. It shows the status of the ports and the IPs and all that stuff we've already seen before. All right, so since those are both up, let's run off to the browser. And I already have the IP for the virtual machine here. We're going to do the port of 3000, and this should now hit it, and there we do. Now I'm not going to run the db seeder that we saw

earlier. I'm not seeing any data because this was a fresh container, and so there's no data in the MongoDB database. But we certainly could also run commands against that if we wanted, and that way we could seed it with some data. But you can see it is up and running, and we didn't get any errors there. Now if you wanted to see that hey, there are no errors, then we could also, while we're here, do docker-compose, and we can do logs. And what this will do is give us the logs for all the containers that are associated with this docker-compose that we ran. All right, and there we go. So notice that now I have the entire kind of log infrastructure here, if you will. I'm going to go ahead and exit that out. And you'll notice I can get to all the details about, in this case, the MongoDB setup. Here's my npm it looks like. Here's the calls that went in as we hit the web page. And it looks like, other than me aborting down here, everything is looking pretty good. Now to bring these down, which you've already seen, we can do docker-compose down. And this will go ahead and bring these down, and now we'll be kind of off and running, and we can rebuild the images or do what we want. And I can even remove all the images if we wanted to. When you do the down, there's a --rmi all that I showed a little bit earlier, and we could do that as well. So now let's do docker-compose ps. You'll notice nothing is running there. And we could even do the normal docker client ps -a if we wanted, and there's nothing there. But if we go to docker images, you'll see that I still have my two images here. There's my mongo and there's my node image. So that's an example of how we can use Docker Compose to very easily, not only build, but also run our services and then take those down when we're done. And then if I wanted to set up volumes and all that kind of stuff, we certainly could. We'll see that coming up in the later demo in this module. Now before we wrap up this section, let me show you one more Docker Compose YAML file. Now this is for ASP.NET Core and PostgreSQL. Now again, I could come into our Dockerfile, and we could manually run the different commands that we looked at earlier in the course. But now I have my version, my services, and I have a web and a postgres. And you'll notice that for the web, it's very similar to what I just showed for node. We have a custom Dockerfile, we set the context, we give the ports, and we hook it into a bridge network. And for postgres, it's really close to the same thing. But you'll notice right here, I'm actually adding on environment variable value. Now this is something that the PostgreSQL image knows about. And so we did this manually earlier when we ran docker-client run, we had to put this on the command line, And you could do that, but again, I don't really want to type that over and over and over or copy and paste over and over and over. So we just simply use this environment property. Now I can put in the different environment variables that I'd like to set, and you can certainly put more than one if you'd like. And then we link that into the same network, and we can call this from the ASP.NET Core container that'll be running based on this name right here. All right, so we can do the same exact thing now. If I just had grabbed this, which, actually I don't have this going as an image yet, then we can come into the folder, which I'm already in. We can say docker-compose build. Now this will have to build the ASP.NET image, which will take a moment. And you can see there's a postgres image as well, which I already have locally. So now we're kind of ready to go. And if we do docker images, we should now see that we have the web one here for the aspnetcorepostgresqldockerapp. All right, so that's kind of the first step. Now we know we can do docker compose up. This should now start those up. Didn't do -d, But that's okay, in this case. There's all our logs for that. Now we can come back and hit port 5000. So let's go ahead and try that. Change that up. And we'll be off and running here, and this now hits it. And this one already seeded the database. So this one you can see is definitely working because it shows us the seed data that ASP.NET put in. And if I scroll up, you'll be able to see some of the SQL statements and things. Even the inserts for the seeding are shown here. So now we can break out of here, and I can say

docker-compose ps. There we go. They're both up. And then, of course, docker-compose stop if I wanted. That would just stop them, but we're going to do down, and that'll actually stop them and remove them as we talked about. So that's an example of how you can get started with Docker Compose, with Node and Mongo and a YAML file and then ASP.NET Core and Postgres.

Setting up Development Environment Services

Now that we've taken a look at how you can work with Docker Compose YAML files and some of the different commands you can run, let's walk through setting up a more robust development environment, talk about some of the different Dockerfiles involved, and then look at the custom YAML file and how we can actually run it. So earlier, I talked about this type of environment where we have Nginx on the front end. It can then route to multiple Node.js processes that could be running. And then Node can integrate with MongoDB and cache some data in Redis. Now obviously, in a development environment, you probably don't need multiple Node instances. But I'm going to show you how you could do it just so you see the setup and how it all works. So let's go ahead and jump over to a code demonstration here. And what I'm going to do is, rather than typing it all out, because we've already seen the YAML file, we've seen Dockerfiles, I'm going to talk through the setup, walk you through the basics in this particular section. And then in the upcoming sections, we'll look more deeply at the YAML file and then start to run it and get this environment going. And you're going to see it's actually extremely easy to get going. That's what's so exciting again to me about Docker. So let's jump on over here. All right, so this is a project that has all these different services that I need to have in place, the Nginx, the Node, the Mongo, the Redis. And so what I've done here is I have a .docker folder. Now this is my own name. You can certainly choose whatever you want here. But inside of it, I have my custom Dockerfiles. So I have one for Mongo, for instance. We'll talk through the basics of this. Here's Nginx. Here's my Node, and then here's my Redis. Now these are pretty standard. There's a few things I'll point out and call to your attention. But out of the box, I'm really just grabbing from the latest Mongo. I'm running a couple custom commands. It turns out that the debian:wheezy image of this is based upon, didn't have a particular feature I needed, at least at the time, which is at the time I'm recording this, and it's called netcat. So I had to actually do an apt-get, which is one way on a Linux machine that you can go grab different tools and download them dynamically. So I'm going to do that. And then I'm going to run some custom Mongo scripts and copy them in. Now, in this case, when Mongo runs, I need to supply some username and password-type information. And out of the box, you don't get that. You can supply some basic stuff, but I need to supply obviously my admin password and username. And then the web that's going to hit it needs a web account. So the Node application needs to be able to call it with a specific account as well. And this is all done with some different sh or shell scripts here. And so you'll notice I'm calling this run.sh. And in a nutshell, what I'm doing is kicking off a little bit of scheduling for backups, which, again, in the dev world you probably don't need. But in this particular case, I could use this in a production mode if I wanted. And then I kick off some other things, like this first run. And this is where I use environment variables. And this is how, in this case, a shell script how we can get to some environment variables that are being set. Now these are going to be loaded through this mongo.development.env. And so you'll notice in here, this is an environment variables file. You're going to see this as we get into the Docker Compose file, which is back down

here. But I I can supply the environment. I can supply the Mongo type of functionality for the username and password. And these scripts take care of applying all of this information to the actual MongoDB database. So that's really what this guy does. The entry point runs this custom script, and that kicks off getting the username and passwords all updated in the database. That way, I can truly do authenticated calls from the web app. Now the Nginx also does a little bit of extra stuff. First off, I have a configuration file. And if you're not familiar with Nginx, it's a reverse proxy. And it's something that, as was shown in the diagram earlier, it could be hit first on port 80 for instance. And then it could forward dynamic calls that Node needs to handle or whatever your server is on the back end to, in this case, the Node process. But for the static resources, and this would be your CSS, your JavaScript, your images, things like that, I really don't need to hit a back-end process for that. Why not just let a really efficient server, like Nginx, serve those up? And that's what I'm going to do. So this has the configuration for this proxy server, and so it's actually going to configure a few things. If I go into config, nginx, you'll notice in here if I scroll on down, we have this little node-upstream. And don't feel like you need to understand this if you're new to it, because, quite honestly, you could just go to their documentation, copy and paste some samples, and tweak them. But I wanted to just point out that I actually set up a node1, node2, and node3. So when a request comes in to Nginx, if a dynamic call is required, in other words it wasn't a static resource like a CSS file, then it can route it into one of these Node instances. Now as mentioned earlier, it's not like you need three Node instances in development. But maybe you want to simulate a production type of scenario and do some testing. Well, it's pretty easy to do that. That's kind of why I set this up. So that's really all I want to point out. You'll see that these are running on port 8080, and these are going to be three containers that ultimately will run behind the scenes. Now the rest of this, if I go back to the Dockerfile for Nginx, is I actually copy the public resources. This is the static resources. If we jump over here to public, css, img, and js. So I'm actually copying that up into the container that's going to run for Nginx, and that way it can handle serving that. I do a little bit with certificates in case you want to play around with SSL. These are self-signed certificates so they wouldn't be used for production. And then I kick it off down here by running the Nginx command that you see right here. So that's that Dockerfile. I also have one for Node. This one's pretty basic. It does a little bit of an install as far as npm install, installs something called pm2. This is a process monitor that'll monitor our server process for all the Node instances. And if the server process dies, it'll restart it. Or if I change the code, it can restart it. And that's a real nice thing to have obviously in development mode. And then finally, I have the Redis image. And really all I'm doing here is copying again some configuration file info, and that again is located up in this config. You'll see Redis. And all I'm doing here is supplying a password for the caching server. Don't use that in production, but it's not bad for dev. It's just password. All right, so that's a quick runthrough on the services that we're going to have running and the Dockerfiles that are going to drive these images and ultimately the containers. And by using these, you're going to see that we can make a Docker Compose file, and we'll be doing that in the next section here, and then get that up and running very quickly and efficiently using the Docker Compose commands.

Creating a Custom docker-compose.yml File

Now that you've seen the custom dockerfiles that are going to drive the services that we're going to get up and running in our development environment, let's jump into the

docker-compose.yml file and see how it's used and the different services that we have in it, and some other features, and see how we can create that. So earlier we looked at the custom dockerfiles, and we saw that we had our mongo, nginx, node, and redis. Let's jump on down to the docker-compose.yml file. Now to start things off, you'll notice that I have the standard version up top, and then I have my services. So from a high level, I have my nginx, a node1, 2, and 3, we have mongo, and we have redis for our caching, and so that's the same infrastructure that we talked about a little bit earlier in this module. Now let's walk through each of these real quick and just take a look at what's going on with our individual services. So first up is nginx, you can see, and it has a container_name, so there's a property that you can put in your YAML files called container_name. We've already seen the build context up to this point in the modules, and we're setting the build context as this folder here, the root of this folder, which is CODEWITHDANDOCKERSERVICES. And then you'll notice that I'm pointing the dockerfile location to that .docker folder that we looked at earlier, and of course the actual dockerfile. Now, if you recall, in the config for nginx I configured node1, 2, and 3, and that way, when a request comes in it can kind of load balance and does a round-robin by default, and it'll call node1, and the next request goes to node2, and so on, and so forth. So what I have here, if we go back to the docker-compose, is those actual node1, node2, node3. So this is going to link up to these services here, and this is kind of like an alias, and then it points down to the node1 definition down here in 2 and 3. So those are actually really important in this case because of the nginx acting as a type of load balance, or reverse proxy actually, and it does that type of thing. And I'm also exposing the ports that I want nginx to support. Now, in this environment for development, I'm probably just going to hit port 80, but I showed earlier that in the dockerfile for nginx, I actually do load up some self-signed certificates, so it would be possible with some more configuration code to get SSL going if we wanted on port 443. Now, the next thing I do is I load an environment variable. Now, this environment variable, it's just one, you'll see, but it's in a file, this .env. So let me show you what this file looks like. It's very, very basic, but really, really useful. So you'll see this app.development.env. And all I'm doing is adding this NODE_ENV and setting it to development. Now, normally, that's used just with Node.js, but I may actually want to use that particular environment variable throughout multiple containers. Now in this case I'm not really using it, per se, but it would be available. So what'll happen is when we build and then run this, it's actually going to load that environment variable and make it available in the container so that we can work with that if we'd like. Now you might wonder what this is. Well, I'll show a little more of this as we get into the next section and actually run the docker-compose commands, but I have a little kind of README up here of what to do to get this running. So the first kind of step after you do some other changes for connection strings is I have to export app environment, and I set it to the environment I want to run. Now, right now I only support development. You'll see I don't have an app.staging, app.production, mongo.staging, mongo.production, just development. But as I'm getting ready to migrate this to other environments, I can certainly add those files. And I just made kind of my own way of doing it, a little environment variable that's local to the particular file here, and it will be read dynamically. So we'll run this in the console, and then when we do the docker-compose build and the docker-compose up, then it will automatically make this available. So this would load app.development.env, assuming I set it to development, and I'll show you this as we move into the next section with the command line stuff. Alright, and then the final thing is I have a custom bridge network called codewithdan-network, and that's, again, going to allow all these containers to communicate in the same network on that Linux host. Now, for the node it's very similar. I have a container_name, I have a build location

for the dockerfile, expose the ports, but here's where I actually set the volume. Now this assumes we're in development mode, because you'll see that I'm pointing to the local folder, which would be everything you see here. And then on the actual container, though, this is where we do that kind of aliasing, and the volume actually is going to point to my code here. So once these containers are up and running, we can make our code changes. And then I showed earlier how I have, it's called pm2 monitoring, and that pm2 will kind of watch for changes, and if anything changes then it'll reset the server.js, and that way I can just leave my containers up and running and they'll reset each other, or themselves, I should say. I set the working directory, and then I also load an environment file. Now, this is used because this is Node, and again, the environment file is specific to Node. And so Node and Express, specifically Express is being used as the web kind of component of this, it knows how to read that environment variable, and it can actually tweak some settings there. Now you can have multiple environment variables. This is something that if I wanted I could have Env1 or Env2=foo, or whatever, and just keep going, name-value pairs. And so this makes it really, really easy if you have a bunch of environment variables, well, I could put these right in the yml file here. It's a lot easier just to put them in an environment file and then have them loaded up, and then, as mentioned, if I do this export APP_ENV=development, then what'll happen when I use this with docker-compose is this would be app.development.env. Now, obviously, if I change this to production, then it would be app.production.env, and that would help you kind of dynamically load the different environment variables. So this is the same for all the node containers, and again, I put three of them, mainly because you might want to simulate your production environment while you're in your dev environment, and so this is just kind of allowing you to do that. Now, if you only wanted one, you could certainly delete, you know, two of these. Now, the mongo is actually pretty simple. We have, again, the container_name and the build. Here's the ports, external, internal. And then I have environment variables, and these are really important, because the mongo one, this is going to be available to the container, and then the shell scripts that I showed a little bit earlier, they're going to read from these environment variables, apply them to Mongo, so you'll see that I have my kind of root admin account, and then my webrole here represents what Node would use to actually call in the mongo in that network. And so what'll happen is the shell script would read these as they're passed into the container once it gets up and running. And then once it's done applying those to mongo, it would just erase those out of memory, because obviously, once mongo is up and running and we've configured it, we don't really need those or probably don't even want those hanging around in the environment. So that's one way you could do it, there's certainly other ways that you could configure this, but this makes it easy for a development environment. Alright, and moving on down, there's the environment variables being loaded. Then the last one is redis, very similar procedure. We have our dockerfile. This is the redis port that's kind of the standard port. We load that environment variable. This one just loads the node, which isn't used here, but I could have environment variables that are maybe specific to Redis potentially. And, again, we put it in the network. And so there you have it, that would be the entire file we need. And what I love about this is right now we have six services in here, nginx, three nodes, redis, and mongo, but if I wanted to add a seventh or an eighth or a ninth service, then I could certainly do that just by updating this file. And then you've seen how easy it is to bring services up and take them back down, and that's what we're actually going to talk about in the next final part of this module here. So that's an example of the custom YAML file that could be used to get our development environment up and running. And this is something we could just check into

source control, every team member could pull it down, and then we would be able to do our builds and start running our containers, and that's what we'll take a look at next.

Managing Development Environment Services

At this point, we're all ready to go. If we have that YAML file local with our source code, now we can use the Docker Compose tools, and that's what I'm going to walk you through here to get this development environment and all of these services up and running in just a matter of minutes, especially if you already have some of the images already cached locally. So let's jump in and take a look at how we can do that. So I already have a Docker Console set up here ready to go. And all we'd have to do as we've seen earlier is run docker-compose build. Now I already have some of this cached to kind of speed it up, but this will go through and build out our different services. We have six of them again. And now that these are built, we can do a docker images, and you can see that, here we go, I have my NGINX node1, 2, 3, and these are some other ones that I had. But you can see here's the codewithdandockerservices_node1, 2, 3, Redis. But the reason this built pretty fast is I already had another version of this going, and so I was able to leverage some of the layered file system. All right, now if I run docker ps -a, you can see that I've already tried to run some of these before. And so a little trick we can do here is we could say docker rm, and I'd like to go ahead and remove all these. We're going to start kind of from scratch here. And when I do a rm, I can do a -f to force. And that way, if anything is kind of locked up at all, we can take care of it. And now what I'm going to do is say docker, and we're going to list all the images, ps -a and -q for quiet, and this will go through and remove them. So let's do docker ps -a, and let me do that one more time. There we go. Nice and clean. Okay, so just to kind of show that we're starting from scratch here on our containers. So now it's pretty easy. You already know what to do. In fact, we've done this. We can do docker-compose up, and let's go ahead and run this. And you can see it's now bringing up all my different services here. Mongo's loading. Here's some of my Node images that are loading now. They're creating some routes behind the scenes to handle all that. Db connection, you'll see, is opened. If I scroll up a little bit to the Mongo section, you'll notice that it's actually showing me that a root user was set called dbadmin and a root role was set as well. And then we have a webrole and a database name and all that stuff. And I'm just logging that out right now so we can see if it kind of worked. And so you could scroll through all the logs if you want. I'm in this case running in interactive mode. But, of course, I could have just done docker-compose up -d and then run in the daemon mode that you've seen. And now that this is up and running, we can come on over, and I'm just going to refresh. I've already run this. And there we go. It looks like the content's been loaded here. Now I'm not seeing any data, of course, because at this point, I haven't run the seeder. But since we have containers, I could go on back again, and we can kind of x out of here. Now notice it's going to try to gently shut down our different services, so we'll go ahead and let it finish. All right, and let's do a docker-compose ps, and you'll notice here that they've all exited. You can see that over here on the right. Okay, that's fine. We know how to do the up. So let's go ahead and do the up again. But we'll do -d this time. All right, that'll let me get back to here. Now you can see the names are actually shown right here. So I just need to do a docker exec, and then I can put the name of one of these containers. So let's go ahead and grab this guy as an example. And I can execute that node dbSeeder. I showed this a little bit earlier in the course. This is a file that's in the actual project that will get some fake data up into Mongo. So we can try to run that. It looks like it

worked, so we'll exit out of there. Again, do docker-compose ps. All right, it looks like everything is up and running. You can see the state right there. And let's refresh. All right, there we go. Now we're getting some data. This is all from the database and this. And, actually, now it just cached in Redis, so this data right here is being cached because it doesn't really change much. So every time I refresh, it actually is going to be pulling from Redis. So we could actually do a docker-compose logs, and we can get back into the logs. And you can see some of the Redis connections and things going on here. So that's an example of how easy it is now to get this custom, six-service development environment up and running and allow us now to have a fully functional website. I can start editing the code because of the volumes because, remember, we had a volume that points to my local machine in this case. And now when I'm done, I can just close up shop for the day if I want. We'll get out of the log mode here and do, like we saw earlier, we could do docker-compose down. And then that, of course, will stop the services and, as you saw earlier, also remove the containers. So there you have it. There's a walkthrough of setting up a custom development environment with six services all the way from looking at the custom Docker files, some of the configuration environment files, to the YAML file to actually running it with our docker-compose commands.

Summary

Docker Compose provides a great way to manage the process of building services and then starting and stopping those services. And, of course, we talked about how behind the scenes really a service is a running container. Now, of course, starting and stopping services by hand, using just the command-line, is a little bit challenging when you get more than one or two. So we talked about there's a docker-compose.yml file that defines all the services, and it's an excellent way to manage these services. It's very easy to write, allows you to define custom networks like bridge networks, define ports, environment variables, and much more. And then, as we talked through this module and looked at our Docker Compose files, we also talked about some of the key Docker Compose commands, such as build, up, and down. And then, of course, there's others like ps, to view your running services, and you can call start and stop and things like that. I personally think that from a developer standpoint, understanding the fundamentals of Docker Compose is really, really important, especially if you want an easy way for you, or maybe even a group of people on a team, to very easily have a consistent development environment that could also be deployed into staging and production environments. It provides a very productive way to do this, of course, as you saw, and it really takes a lot of the headache out of the picture that we've traditionally dealt with in the world of software development and all these different services that we often need. So I hope that gives you a great fill for the power of Docker, and once we combine Docker Compose with our images and containers, you really can do a lot with just a little effort.

Moving to Kubernetes

Introduction

Up to this point in the course, you've learned about images and containers and even seeing how you can orchestra multiple containers using Docker Compose. And that works great while you're on your development machine, but what do you do when you're ready to move to another environment? Well, that's we're going to look at in this module. So we're going to look at movie from Docker Compose to something called Kubernetes. Specifically, we're going to cover the following topics. We're going to talk about how Docker Compose is great for some things, but not so great for others, and we'll talk about what's missing there. I'll introduce you to Kubernetes and give you some fundamentals on the basics of what it is. We'll then talk about how we can convert from a Docker Compose file to some files that Kubernetes can use to run your containers. We'll then look at commands you can use in Kubernetes to actually get your containers up and running and also how you can stop and remove those containers. So let's go ahead and get started.

Beyond Docker Compose

Docker Compose provides a great way to get multiple containers up and running on your system, but what do we do as we're ready to move to a different environment? For example, you might have a staging or production environment, and now we want a very robust way to run our containers. Well, up to this point, we've seen how we could just run Docker Compose up and get different containers going, such as nginx, different APIs, even databases if we wanted, and something like Redis if we were using that for caching. But as we move between the different environments, how are we going to manage our containers? Because with Docker Compose it's going to be more manual, somebody has to run those commands or program something to run those commands, and what we do is we want to scale and do other types of things. Well, Docker Compose actually does have a scaling feature, but it's not designed for things like load balancing. Docker Compose has a policy where you can restart containers if they fail, but that's about it. Anything you want that's more robust for production, you're kind of on your own. In addition to gracefully handling containers that may fail, what if you want to scale and load balance your different containers? Well, Docker Compose does provide some scaling features, but it doesn't provide any load balancing features. So if you want to run your containers on multiple VMs and then load balance between those nodes, if you will, then Docker Compose isn't going to help you there, and that might be a big deal depending on what your production needs are. So wouldn't it be nice if we could do the following things across our different environments? First off, package up an app, provide a manifest, and let some other tool manage that for us; not worry about the management of containers; eliminate single points of failure, and even self-heal containers if they have a problem; have a really robust way to scale, but not only scale, also load balance across our different containers. What if we could update containers without bringing down the application and even have some robust networking and persistent storage options? Well, all of these are obviously good things to have, and Docker Compose does quite a bit, but again, it was never really intended to be the production environment, even though you absolutely could use it if somebody had set up the commands to run across your different

environments. So what if we could define containers we want, hand it off to a system, and basically tell that system, hey, here's what I need, now make it happen, you manage it. Well, welcome to Kubernetes. That's what it can do for us, and that's what we're going to start looking at next.

Introduction to Kubernetes

So what is Kubernetes? Well, that's actually a very large question to answer, and that's why there's multiple courses on Pluralsight about Kubernetes. The goal of this section is to gently introduce you to the basics of Kubernetes and some of the key features that it offers. And then through the rest of this module, I'll walk through how we can move from Docker Compose into Kubernetes. So if you go to kubernetes.io, you'll see this on the website. "Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications." Now the way I like to think of it is from the perspective of teams. If you're into sports, then you could think of each player on the team as a container, and Kubernetes would then be the coach of the container team. Or if you're not really into that, and let's say you like orchestras, well, Kubernetes would be the conductor of a container orchestra as you see here. You may look at that and go, well, isn't that really what Docker Compose does? And yes, but you're going to see Kubernetes offers a lot more than just being the conductor. So here's a quick overview of Kubernetes. First off, it's designed for container and cluster management. Now we'll talk more about clusters in just a moment. It's also supported by all the major cloud platforms out there, and that's a big deal if you want something that's very standardized and also very popular to work with. It provides a declarative way to define a cluster state using YAML files, manifest files they call them. So it's very similar to Docker Compose in that regard. And then, finally, it provides a command line tool that you can use to interact with the Kubernetes API called kube c-t-l, kube cuddle, however you'd like to say it. I've always learned it as kube cuddle, but if you prefer kube c-t-l that works as well, and we'll be talking about this command as we move forward. Now, some of the key features then of Kubernetes that really set it apart from something like Docker Compose would be the following. First off, it has load balancing and discovery of different services. And we'll be talking about what a service is a little bit later. It can orchestrate your storage, handle automatic rollouts and rollbacks, and manage different workloads. Really, in a nutshell, what it does is you give it a final destination that you'd like to get to, and it's kind of like the mapping software that shows you how to actually reach that destination you'll see. Now it has a few other things, like self-healing of containers. It can manage secrets and other configuration data. Horizontal scaling. And this is not just of containers, but you can also have multiple nodes, they call them, which is your virtual machines. And then we could go even deeper. It has a lot of other features with networking and more. So from a high level or big picture view, this is what Kubernetes looks like. It's going to have a master node, and the master is in charge of keeping all the children in line, if you will, and we call these the worker nodes. Now, the worker nodes would have something called Pods inside of them. Now a node is like a VM, and a Pod is a container for containers. So think of one of those shipping containers you might see on a boat, or maybe you've even had one delivered to your house, and you use it to move. Well, you can think of that as a Pod, and then you can put multiple things inside of that Pod, and multiple Pods could even run on a different node. Now, with multiple nodes being managed by a master, we call that a cluster. So we can have a cluster of virtual machines. We can scale out. We can scale in. We can scale the individual

containers out, the nodes out, and even control that based on different scenarios, such as, maybe the CPU load is too high; therefore, we want to automate a new node being created or maybe just different containers being started or stopped. So now that you've learned about the fundamentals of Kubernetes, learned a little bit about the master, the worker nodes, Pods, and how containers can run in Pods, let's take a look at how we can get Kubernetes running locally on your machine.

Running Kubernetes Locally

Getting Kubernetes up and running locally is actually a fairly straightforward process, and there's multiple options you can choose. One option is called minikube. Now this does require some setup work, and you'd need to run off to the GitHub site shown here and follow those directions to get it up and running. Now the other option is what I'm going to be demonstrating, and that is Docker Desktop, formerly called Docker Community Edition. Now Docker Desktop has Kubernetes support built in out of the box. So it's going to be as simple as checking a checkbox, which I'll show you in a moment, and you can have Kubernetes up and running on your machine. Either of these options would work, and if you have the luxury of being able to run Docker Desktop, that'll certainly be the easiest one. But minikube might option if you need to go that route. So now that we've talked about that, let me jump over to the Mac side and then the Windows side, and I'll show you how easy it is to enable Kubernetes support in Docker Desktop. So on Mac, you can come up to the Docker whale icon, and you'll see that I already have Docker Desktop and Kubernetes running. But if I go to Preferences, this is where you can enable it. So along the top tabs, you'll see Kubernetes. And all you need to do is come on in and check the checkbox, Enable Kubernetes, and then hit the Apply button. Now I'm also going to show that there are some options available if you want to just use a Docker Compose file with Kubernetes. And to simplify that, you can check this checkbox, and I'll talk more about that in an upcoming section of this module. Now on the Windows side, we can come down into the tray, find our Docker icon, and then right-click and go to Settings. Now just like on the Mac, you're going to have a Kubernetes option. You'll notice it's definitely a different arrangement, but we can click on that. And the same thing goes here. We can check Enable Kubernetes and then hit Apply. And then, if you'd like to use Docker Compose files to run Kubernetes as well, then to enable that option, check the checkbox here and hit Apply. Once you enable Kubernetes, it does take a little bit of time for it to fire up the first time, but then you can just leave it up as you're doing your development. And now what we're going to do is talk about different options for getting from Docker Compose into Kubernetes.

Key Kubernetes Concepts

Before we officially convert between Docker Compose to Kubernetes files, let's talk about a few key concepts that you need to know in Kubernetes. Now, as I mentioned at the very beginning of this module, Kubernetes is actually a very big topic, and that's why there's full courses just on Kubernetes. The goal here is really just to break you in gently, if you will, to some of the different concepts that you need to know so you can get up and running quickly. But know that there's definitely going to be more research involved on your part if you want to use this in a real-life scenario. So the first thing I want to talk about that we're

going to see coming up is something called a deployment. This plays a central role in Kubernetes because it allows us to describe the desired state we're after in one of two types of files. We can use a YAML file, which is the normal one, or even a JSON file. So what we'll be doing as we convert from Docker Compose to Kubernetes is one option would be to basically translate the services in our Docker Compose file that define the images and the containers that should run into a Kubernetes deployment file. I'm going to show you an easy way to do that to get started. But really, what a deployment is all about, in a nutshell anyway, is saying, hey, I need these five containers up and running, and I need them to be able to communicate somehow. Now in addition to describing the desired state in a deployment, you can also use it to replicate pods, and this allows you to scale out if you'd like, add more pods onto a node, for example. And it even supports rolling updates and rollbacks, which can be very important as you version your app into the future. Now in addition to deployments, we're also going to see something called a service. Up to this point, we've seen that containers run in pods, but pods can live and die. If a container goes down, that pod may be completely removed and a new pod might be brought up. So they could have a very short or long lifespan. It really depends. So from a consumer standpoint, we can't always count on the IP address of a pod and ultimately the container being there. Services allow us to abstract those pod IP addresses from the consumers, and I'll show you a visual of this in just a moment. In a nutshell, though, that's a good thing because now, if a pod dies and a new one's brought up, the consumer doesn't even know anything happened. Everything is good to go because the service kind of acts as the middleman, if you will. In cases where you have multiple pods that a service knows about it can also load balance between those pods. And that could be good, as we talked about, when it comes to scaling out and those types of scenarios. So here's a visual of how this all works. A service is going to have an IP address, but a pod is also going to have an IP address. Now ultimately, a consumer of this pod wants to get to the container, but if we gave them the 10.0.0.43 address, well, that might go away in the next few seconds, potentially if something happened to the pod. So, what we'll do is instead give the consumer the reference to the 10.0.0.1 IP, which is the service. That way, if the pod changes, or maybe even a new pod gets added with different containers, the consumer just has to remember that one IP and know about it to talk to the different pods. Now behind the scenes, the service is going to be in charge of knowing about the different pods. If one goes down and a new one comes up and the IP address changes, then the service will automatically take care of that. So it's a level of abstraction to allow consumers just to have one endpoint they need to know about. Now are there more files and concepts aside from just deployments and services? Yes, there are, actually. There's a whole bunch of things we could get into. But these are two of the key ones you're going to see in just a moment as we talk more about converting from Docker Compose to Kubernetes.

Converting from Docker Compose to Kubernetes

We've talked about the fundamentals of Kubernetes, how to get it running locally on your machine, as well as some of the key concepts such as deployments and services. So now it's time to actually take a Docker Compose file and either run it directly in Kubernetes or convert it into some of the files Kubernetes normally works with. So there's several options when it comes to migrating from Docker Compose to Kubernetes. One is just to use Docker Desktop directly. It has an open-source project built into it called Compose on Kubernetes, and I'll show you this in just a moment, a very easy way to get started. Now there's another project called

Kompose that you could also use. This doesn't run directly in Docker Desktop. It's a separate open-source project, but it will directly translate from a Docker Compose file to the different deployment, service, and other files that Kubernetes normally needs. And this is a good way to go because, as mentioned in a previous section, if you're working with a DevOps team and you're just trying to get Kubernetes running locally just to make sure your app runs as expected, but then you want to be able to hand them off some of the work you've already done, if they work with Kubernetes, they're probably going to want the different Kubernetes files. So let's take a quick look at both of these and how we would get started with them. I'm going to start with Compose on Kubernetes. So I've gone to the GitHub site that Docker maintains for this. And as mentioned, this is actually part of Docker Desktop now. So you have this already if you're running Docker Desktop. Now if we scroll on down, they'll give us an idea about what we're going to be doing, and they call it deploying a stack. When it comes to clustering and having multiple VMs or nodes out there, Docker has their own solution called Docker Swarm. Now they also directly support Kubernetes, of course, as we've already seen, but they have a docker stack command that's been in there for quite a while now if you want to get a swarm or a cluster setup. Now what they've done is made it so their normal docker stack command can actually be used to deploy, but using a Docker Compose file. It's actually very simple. So, assuming you had a Compose file like you see here, we can run a docker stack command. We could deploy it using the Kubernetes orchestrator, and then give the Compose file, which is docker-compose.yml, and then give the stack of name. Now earlier I mentioned when you come into the Preferences on Docker Desktop and go into Kubernetes, there's this checkbox, Deploy Docker Stacks to Kubernetes by default. Now, if you enable that, then when you run these docker stack commands, you won't have to set the orchestrator. It'll just automatically do that for you. So that's an option if you'd like to go that route. Now if you don't want to work with Docker Compose for Kubernetes and instead just want to go 100% Kubernetes, then we can go to the Kompose project. Now Kompose actually translates and converts your Docker Compose files into the deployments, and services, and persistent volumes, and other types of files that you can have to work with Kubernetes. To get started is really easy. You come on into the Installation, and you can see for Linux and Mac, they have some curl commands, and for Windows they have some instructions. You can download a binary, and then add it to your path. Now I've already installed Kompose on my Mac and run these steps here, so it's ready to go, and I'm going to show you how we can use it. But let's go to the get started guide. So if we scroll on down a little bit, you're going to see a kompose convert command. And what it will do is take that docker-compose.yml file that you'd have wherever you run this command and automatically generate the different service and deployment files for each of the services that are in your Docker Compose file. So if you add four services, then by default you're going to get four service Kubernetes files and four deployment Kubernetes files. And you can see that kind of here. Now just like Docker Compose, there's even a kompose up you can run, and that will allow you to actually run directly without really having to worry about the different files. I personally like to have the files because oftentimes I need to tweak them a little bit. But this gives you a good starting point with a convert command to do that. So let me show you that real quick. So if we come back into a project, I've opened up the CodeWithDanDockerServices project that you've seen several times up to this point. And if I come on in and type kompose convert, if I were to hit Enter, that would take this Docker Compose file and convert it into the deployment and service files I mentioned. Now I've actually already done that for the project. You're going to find a Kubernetes folder. K8s is an abbreviation for that, by the way. And then here's what it generated based on what it found in this Docker Compose file. So I had a mongo-service. It

did all those deployments and services, nginx, node, and redis because that's what we had in here. Now in order to get this going, if you want to run it on your own, there's a README me at the route of this project that'll help you out there. But you do need to run this in production mode, which means exporting or setting if you're on Windows, the app environment to production, set in your DOCKER_ACCT, removing a node volume, and you'll see that right down here in my node service. And then after that, you'd be ready to go. You can do a docker-compose build to get your images. Now we could come in and do our kompose convert, or if you just want to have one file instead of many, you could say --out, and I'll just say test.yml, and I'll show you the output here. So now if we go to the route, it just generated this file. And you'll notice that we have a lot of different YAML code in here. In fact, it's pretty large, actually, with everything that we want. Now I normally like to work with the individual files, as I mentioned, so let's talk through these just really quickly and see what a deployment and what a service actually looks like. So if we go to the mongo-deployment, if we come on in, it'll say how many replicas we need. Well, we just want one of these database items in this case, one container. It's going to say some environment variables here, some key value pairs, and if we come on down, you'll notice image: danwahlin/mongo or whatever you called your image, in this case. Now, if I go into the mongo-service, the important part is the ports down here. Now as we talked about earlier, a service provides an IP and abstracts the consumer from one or more pods running behind that service. In this case, there's just one. It would be for mongo. But this will define how to communicate to that pod through this port. We have an external and internal port. Now the same goes for nginx.. It has a deployment. It also has the image. It has a service that has our two ports in this case. And if I go down to node and the redis, you'd see the same types of things. The deployments define the image to run for the container, and then the service defines the ports and can have other information. Now that's a really quick look at what we can do because with services, there's different types of services and ways you can use them. But this will be enough to help us move from our docker-compose file here to actually Kubernetes files. So now that those files are generated using the Compose open-source tool, let's take a look at how we can get Kubernetes going and get our containers in Kubernetes.

Running Containers in Kubernetes

Now that we have our Kubernetes deployment and service files available, it's time to actually run our containers in Kubernetes. And I'm going to show you the process, as well as a few key commands to make that happen. So before jumping back to the command line, let's talk about a few key commands that you're going to want to know if you start working with Kubernetes. Now as with Docker, there are a lot of commands you use every day, and there are some you just don't use that often. So these are some of the commands that are very frequent, and there are certainly a lot of other commands you could use as well. So first off, if you just want to get the version that you're running, you can run kubectl, or kubectl, however you like to say it, version. If you want to get information about deployments or services or Pods or nodes, then you could do the get command. We can also get one container up and running very quickly with a run command, and this allows us to name what it is we're going to do and then pick the image. If we have our deployment files and services, which we have, then a very easy way to get all those going is to do an apply. Now with apply, you could say -f and either give it a file name of, for instance, a deployment, or you could just give it a folder name, and all the different Kubernetes files in there would be applied. I'm actually going to be

using the folder name trick here in just a moment. And then, finally, as you're working on your local machine, you'll probably want to expose one of the Pods so that we can actually get to it. So we can do that through a port-forward command, and I'm going to show that as well. So now that we've seen some of the basic commands, let's see how we can use a few of these to actually get some information about Kubernetes and then also go in and run our containers. So coming back over to the CODEWITHDANDOCKERSERVICES project, I've opened that up, and I have the Kubernetes files I mentioned earlier available here. So we have, again, our deployments. That gets us to the desired state we want to get to for Mongo, NGINX, Node, and Redis. And then we have the services, which, again, acts as kind of an abstraction layer for IP addresses so we can talk to those Pods. All of those do much more than just that. But I want to emphasize that because there's quite a bit to it, but this will be enough to get us going. So the first thing I'm going to do is we could come on in and just run the kubectl command for the version. And this will give us some information, probably more than you want. But a nice reason for running this is it lets us know that, yeah, things are working correctly, and the kubectl command is actually giving me back some info. Now we could also come in and say, Hey, Kubernetes, what deployments do you have? So we can get deployments, and right now there are none. We could also run this and say, What Pods do we have? Well, likely, if we don't have any deployments, we may not have any Pods, although it depends. No Pods found. We could even come in and say, What services do you have? Now we do have one here you'll notice. This is the cluster IP address for the overall Kubernetes cluster on my local machine. Now as a heads-up when it comes to the cluster, when you're working with Docker Desktop on Windows or Mac, you're going to get one node in your cluster that you can use. So you're not going to able to scale out the actual nodes in this environment. But it's just enough that we'll be able to play with different flavors of containers, get them orchestrated, and test everything out. So it's very good for testing purposes to try out your app in Kubernetes. Now if we wanted to just get a single container such as the NGINX Alpine up and running, then we could say I'd like to run, I'm going to call it nginx-server, and then give it the image. Now, in this case, I'm going to pick nginx:alpine mainly because it's pretty small and quick and easy. And then when we run this, if I hit Enter right now, we're probably going to get a message saying that a particular generator template is deprecated. It'll tell us another command we could run. Now I fully expect into the future that will probably change, but as of today, that's what I've been getting. So let me go ahead and run it. Okay, and there's the message. So right now it's using this apps.v1 generator template, saying it's deprecated, and that we should instead use --generator=run-pod/v1. Now, in this case, this is good enough for our demo though. If we go in and say, Let's get the deployments. And as a quick little tip here, you don't actually have to type it all the way out. You could just say deploy for example. There we go. There's our NGINX server. Let's clear that and let's get any Pods. All right, and there is the name of our Pod. Now, currently, the container inside of this Pod is running on port 80. But if I run off to the browser, and let's refresh here, you'll notice I can't hit anything. Well, that's because it's not exposed outside of the cluster, so you can think of it, it's internal to the cluster currently. Well, this is where the port forwarding is useful, especially for just running things locally. So one more time if we run kubectl get pods, this name right here is important. So I'm going to copy that. And now I'm going to do the kubectl. We're going to run port-forward, give it the name of the Pod. So I'm going to paste in what I just copied. And then, externally, just so I can show you it's working, we're going to say, externally, use 8080, and forward that to 80 on this Pod because that's what NGINX is running. Now this is going to lock up the console the way I've done it. But that's okay. You'll notice a forwarding message comes up. And if we refresh this,

obviously this isn't what we want. We're not on 8080, but let's try out 8080. And there we go. Now we're able to hit our first Pod with a container inside of it, which is NGINX Alpine. Now I'd like to delete that, so I'm going to say kubectl delete, and we have a deployment here, so I'm going to say deploy or deployment if you want, and then nginx-server. Now that's going to go ahead and delete that deployment. So now if we say get deploy, you'll notice no resources found. Now sometimes it takes a little bit of time for the Pods though. So let's take a look. Okay, now, that's already cleared out. On occasion, you might still see the Pods, but the status will be that, hey, we're trying to take this down and delete it, destroy it right now. Likewise, let's go back to the services, and you can see we're back to just our cluster IP here. Okay, so we're making some progress. Now the last thing I want to show is how do we go in and apply all these files? Because I actually want to get the real thing going here. Well, we can do a very similar process to what you saw already, but now we're going to use a new kubectl command called apply. And then I'm going to say that the file or, in this case, folder I want to get to is the .k8s that I have. Now what that will do is iterate through these files and apply each of the deployments, the services, and I even have some environment variables here, into Kubernetes. Now let's go ahead and try this. And there we go. You can see it ran through all of those. We'll give it a sec here. But if I clear it and go back to kubectl get, for instance, services, there we go. We have quite a few now. We have the Kubernetes cluster IP, but we also have Mongo, NGINX, Node, and Redis. So that's a good start. Let's go to kubectl get pods. All right, and there are our different Pods. And let's do kubectl get nodes. Now I mentioned you can only have one node, but I'll show it here. There you go, docker-desktop. Okay, now, if I go back to the browser, and we have NGINX on port 80 in this case, but you'll notice again nothing. Well, the reason for that is we, of course, don't have that port exposed. So let's go back to our Pods, and let's grab our NGINX Pod name here, and then we're going to do the port-forward one more time. And now I'm going to do 8080 to 80, although I could do 80 as well. Now, sometimes if you do 80, by the way, you might have to run that with elevated privileges, just as a heads-up, or maybe even just doing a normal port-forward like this, you might occasionally have to run as sudo on Mac. Or, if you're on Windows, you might have to run in the command prompt as an elevated administrator type of command. Let's go ahead and try it. All right, that looks pretty good. And now let's go to 8080. Let's refresh because that's cached. Alright, and there we go. Now the database doesn't have anything in it, so nothing is loading here. But it is working. We can see it forwarded to our node. The node called the database. There was nothing. But if we wanted to see that or just hook into that database now, we can even do a port-forward on that if we'd like. Then we could pull up a tool to update Mongo. So that's an example of applying multiple deployment and service files, as well as using some of these other Kubernetes commands. And this is something that, quite honestly, you just need to practice a little bit. None of it's really super hard as far as the commands. I think the most challenging part is getting the YAML files correct and configured. But, as we've seen, Compose and other tools can help with that.

Stopping and Removing Containers in Kubernetes

Now that you've seen how to get containers up and running on Kubernetes, let's take a look at a final command on how to stop and remove containers that might be inside of Kubernetes pods. Now, earlier, you saw me used the kubectl command to delete, and we deleted a specific item and that particular item was a deployment. Now that led to everything that the

deployment created being deleted as well. Well, in cases where we have multiple Docker Compose services that have been converted into deployments and services for Kubernetes, we can use the same type of command, but we can use the folderName. So instead of kubectl delete deployment and then the name of the deployment, we can actually say hey, here's a folder, go ahead and run everything in there and undo it, delete it, and that makes it very easy to clean up everything. So let's take a look at how we can do that with the existing app that we got running in Kubernetes. So coming back to the console earlier where we did the port forward command, we can go ahead and stop that, but doing that, of course, just keeps everything running. We're just not forwarding that port any longer. So if we do a kubectl and we get the pods, we're going to see all the pods. Now, I could delete each individual pod, of course, but because we already have all the manifest definitions for the deployments and the services defined, we could go ahead and do a delete command that simply points to this folder as you saw. So we can do a kubectl delete and then we'll give it the path to the Kubernetes folder. Now doing this is going to go ahead and run everything that you see in this folder off to the left, but kind of in reverse, if you will. So, in essence, when we first ran this and we applied these different files, we were creating the future, if you will. Now we're kind of reversing time. We're going back to what we originally had. So we'll give it a sec, and now I'll come in and do a kubectl on the pods and you'll notice there are still a few there, some are gone, but they're terminating you'll see in the status there. Now if we come in and get the services, everything is gone at this point, except for the cluster IP for Kubernetes. All the different services we started earlier have been cleaned up. Now let's go back finally to our pods and let's see where we're at here and there we go. Now everything has been cleaned up. So by using the kubectl delete command with a -f switch, that provides a really easy way to get our containers not only stopped, but completely removed. And then, of course, we also saw earlier how -f could be used with the apply, and that makes it very easy to apply deployments or services and get our containers up and running and talking. So that provides an example of some of the different kubectl commands and I hope that gets you started. There is a lot more we could talk about, of course, with Kubernetes, but the goal of this particular module was to move from Docker Compose to Kubernetes, and we've now got that process going.

Summary

Throughout this module, we've taken a look at how we can move from Docker Compose to Kubernetes and saw how Kubernetes provides a robust solution for deployments, scaling, and overall management of our containers. By using some of the constructs built into Kubernetes, we can provide a way to move to the desired state, and we could do that using YAML or even JSON files to represent that desired state and that's done through our deployment services and other options available. We also talked about how nodes and pods play a really central role in Kubernetes. Now nodes were the VMS, whereas pods act as a way to group one or more containers together. We looked at a container running in a pod and saw how we can even do port forwarding to get to that, at least while we're on our local machine for testing purposes. And we looked at several different kubectl commands that could be used to do things such as run a particular container in a pod, apply different deployment services and other options available, delete, and much more. So while Kubernetes is a very big topic and I would highly encourage you to check out some of the other courses on it on Pluralsight

to get the full breadth of what it offers, I hope this modules provided a nice starting point for you to get you started moving from Docker Compose to Kubernetes.

Reviewing the Case for Docker

Course Review

Let's wrap up by doing a final review of the case for Docker and why we want to use this in our development environment. So we talked about that Docker brings many development benefits to our different team members, and even if you're just a team of one, there's a lot of benefits, because you could bring servers and databases and many of the things up very quickly, and then get rid of those if you're done with them. So we talked about how we can bring up web servers and databases and caching servers and more, and bring those up in a very consistent way across team members, even in distributed locations if we needed to, and then how we can even move those up into the cloud if we'd like. And so there's the benefits of the consistency and the fact that how it runs in development is how it's going to run if you move those containers and images into your staging and your production environments. Now, we talked about that the heart and soul of Docker is, of course, Docker images, and that Docker images are used to create our containers. So some of the key tools that we talked about were Docker Client, and this is, of course, how we can work with our images and containers. We talked about Docker Machine, and how we can use that to interact with a virtual box. And then Docker Compose, one of my personal favorite tools, is especially helpful as we need to work with multiple containers and get those all up and running and talking between each other. And then, if you don't want to work with the command line, we also talked about Docker Kitematic, and it does provide a really nice and simple way to get started with Docker if you really just want to jump in quickly and not have to learn all the commands that we talked about for Docker Client, Docker Compose, and others. Now, we also talked about linking our source code from a running container to a local folder that you might have on your development machine. And this is a really, really important part for us as web developers, because we obviously need a way to quickly and easily make changes to code, get that code running up in a container, such as a Node.js or ASP.NET or PHP or whatever it may be container that is running the actual server. And we talked about how we can do that with Docker Volumes, and how a volume can point to a folder that we set up, and that makes it very easy to link our source code into a running container. Now, when it comes to containers, while we can pull many of the containers out there from Docker Hub, we can also create our custom Docker images. And we talked about how we can do Docker files and create Docker files that can be based on an image that's in Docker Hub, and then we can add our custom functionality into that. Now, as mentioned, one of my favorite things covered in the entire course was Docker Compose, and we also talked about the Compose YAML file. And this YAML file is just a great way, I think, to get your development environment up and running very quickly. You could have 10 different services, if you wanted, and not have to bring those up individually using Docker Client. We could get this YAML file in place, and we'd be off to the races and up and running. So that's a wrap on the Docker for Web Developers course, and I hope that you have a really solid feel now for the role that Docker can play in your development environment. I appreciate you taking the time

to listen to the course, and hope you're able to apply this new knowledge into your Web development efforts.