Course Overview

Course Overview

Welcome to the Building and Running Your First Docker App course. My name is Dan Wahlin, and I'm a software developer, architect, and trainer specializing in container and cloud technologies. I'm really excited to talk with you about how you can get started using Docker containers in your applications. Docker is a really powerful technology that can be used to package up, deploy, and run your applications anywhere, from your machine to a company server to cloud services. So is it worth your time to learn this technology? Well, in my experience, absolutely. This course is designed to provide a quick introduction to using Docker containers in your development workflow, and will provide the key fundamentals that you need to get started. Some of the major topics that we'll cover include why you would want to run your apps using containers in the first place, building custom app images and pushing them to a registry, running your app containers, communicating between containers, and even orchestrating container builds and runs using a tool called Docker Compose. By the end of the course, you'll have a solid foundation to get started building, running, and deploying your apps in containers. Now to get the most out of this course, you'll need a basic understanding of Docker concepts and prior experience developing applications. So with that, let's jump right in and get started learning more about building and running your first Docker app.

Setting up Your Development Environment

Course Agenda

Welcome to the Building and Running Your First Docker App course. I'm Dan Wahlin, and I'm really excited to walk you through what we're going to be discussing with Docker containers and how you can actually containerize your first application. Now in this module, we're going to focus on setting up your development environment. But before I dive into that, I'm going to give you a quick course overview, talk about some prereqs, and where you can go to get the code. So what are we going to cover throughout this course? Well, a lot of fun stuff to help you get started containerizing your first app. We're going to kick things off with, as mentioned, the setting up your development environment. And I am going to do a quick overview of what Docker images and containers are just in case you didn't watch the earlier course in this learning path. From there, we're going to dive into how do you create a custom application image? So if you are writing an application and you're writing the code, what do we need to do next to actually get that so we can run it as a container? Well, the first thing we have to do is build an application image, and I'll walk you through that process. You'll learn about Dockerfiles and a lot of other fun stuff along the way. Now from there, what I'm going to do is show you how we can take that image and get a running application container going for your app. So I'll show you the Docker commands to make that possible, how we can check on the container, and things along those lines. Next up is going to be talking between containers. A lot of times you might containerize your application, but you might even put your database or APIs or other types of things into a container as well. How do we talk between these containers? I'm going to show you some different techniques we can do for that. In the final module, we're going to introduce a tool called Docker Compose, which is one of my favorite tools of all when it comes to Docker because it'll orchestrate running multiple containers, but it will also let you build multiple images that can be used for those containers. So it's a big, big timesaver as you're working in the development environment. So that's what we're going to be covering throughout the course. Now as far as the target audience, this is, of course, aimed at developers. So if you're looking to learn more about how to run your application in Docker, then you're in the right spot. Even if you're not a developer, if you just want to see the process, then you can certainly go through this as well though. For prereqs, the first thing you need is a basic understanding of Docker concepts. Now if you did skip the earlier course in this learning path, don't worry. I'm going to have a quick overview of images and containers and what Docker is all about in this first module. You also need to be comfortable using command line tools because, after all, we are going to be running Docker commands on the command line. And then finally, you need some experience building applications. It really doesn't matter though what framework or what language. I am going to have to choose one to walk you through, and we'll talk more about that a little later. But the concepts that we're going to learn throughout this course can be applied to virtually any framework and any language out there. Now the code samples that I'm going to be using through this particular example are going to be based on Node.js and MongoDB. Again, the technology doesn't really matter because the concepts we're going to be covering apply to everything. But this is where you're going to be able to get what I'm going to be talking through in this course. So you can go to github.com/danwahlin, and you can go to this NodeExpressMongoDBDockerApp URL. You can then clone it, download the zip, whatever you prefer, and then you'll have all the code that we're going to be talking through. Now when you're done with this course, my main goal is that, first off, you understand why you would want to do this, where it could be used, and really why it's so exciting to know this because you truly will gain the freedom to run your apps anywhere, on your machine, on your friend's machine, up in different clouds. It really doesn't matter. If Docker is supported in some way, shape, or form, then you would be able to run your app there. I think it's super exciting because I use Docker in everything I do application-wise. And once you learn it, it's just a really, really powerful concept. Now after hearing this, if you say, hey, I just want to dive in even deeper, then what I'd recommend is either jump ahead to this learning path, or you can check out my Docker for Web Developers course, which will go much, much deeper into these different concepts and give you kind of that end-to-end app development and containerization experience. So now that we've gone through the course overview, the prereqs, where you can get the code, and those types of things, let's go ahead and introduce this first module.

Module Overview

In this first module on setting up your development environment, we're going to start things off by reviewing the case for Docker. Not every technology out there needs to be used in every situation, of course. And really, Docker is no exception, although I'm a big fan of it in most cases. So we're going to review what exactly is Docker and what are some of the key concepts? And then we're also going to talk about the key benefits, especially as a developer using Docker. But this applies to even folks that are working in DevOps, for example. Maybe they're just a cloud engineer, and they want to learn Docker. There's a lot of benefits, and knowing about those are really key because there's multiple ways that Docker can add value to not only your development workflow, but also the process of your deployment workflow. Now from there, we're going to review what you need to install from a software perspective to get Docker up and running on your machine. Now I'm going to focus on Mac and Windows in this particular course, but just know that if you are on Linux, you can run the Docker Engine directly, and I'll show you the web page where you can go to. And if you are on Linux, there'll be a download there for you. But for the rest of us that are on either Mac or Windows, we're going to be running something called Docker Desktop. So I'll talk about where you can get that, how you can install it, and we're also going to explore some of the preferences that you can set up so that you have a little more control over what Docker is doing behind the scenes. And then finally, we're going to explore the application that's going to be used throughout this course and the one that I mentioned earlier when I introduced the course and showed the GitHub link. This is a Node.js MongoDB app, and I want to emphasize one more time that technology is not going to matter as much though. What we're going to cover is kind of technology-agnostic, but I had to pick something, so this is what we're going to go with. But if you're using Java, .NET, Python, something else, then the same concepts are going to apply. And that's what's so great about Docker is it works in so many different scenarios. So we'll examine the app, I'll walk you through the different parts of it, and then we'll move on from there. So let's go ahead and jump right in then to the case for Docker.

The Case for Docker

Before I dive into any technology, I always evaluate is this something that the crowd is just saying I should use, or should I really use it? Because there's too many technologies out there that are very niche and you just don't maybe need them, even though everybody's saying you do. So what I'm going to do here is walk you through the case for Docker because I do think it's important before we dive in and spend a lot of time learning that we really take a step back and try to understand, why would we care about this? And is it actually applicable to me and my business scenarios? So a lot of us have a shipping process, maybe still even today that feels like this. You put your code in the boat, you ship it over to the land of staging or production, and you hope it doesn't sink before it gets over there. Some of us may still have FTP processes that are very manual. We may copy and paste code. And even if you do that, you can get it working, of course. But where the challenge, at least that I've found, comes into play is once you do ship it over, what happens if the server is a slightly different version? Or maybe the security is slightly different? Or we're missing environment variables that we needed? We had those on our machine, but not in production. Well, shipping applications the traditional way works. I've done it for years. But Docker provides a much more predictable way to ship applications, and let me walk you through that. So let's imagine that every container on the ship that you see here is a Docker image. Now an image, as you're going to learn in just a moment, is everything you need to be successful to run a part of your application. So, for example, you might have a front-end app. You might have APIs. You might even have a database because you can containerize databases if you'd like. Well, to run the front-end app, let's say we want to use Nginx as the server. Everything I need would be in that shipping container right there. It would be in something called an image. So think of it as like the list of all the stuff I need to run that front-end app, including the server, any security patches, any environment variables, my code, of course, and anything else you need to run that successfully. Same for the APIs. Same for the database. So now if I move the image that, let's say, you created over to my machine, I can run some Docker commands, fire that up, and get a container, and I know it's going to run exactly the same as it ran on your machine because all the ingredients are there. In fact, if it ran Nginx, I don't even install Nginx on my laptop. I just have to have Docker and the ability to run these containers. If we break it down and talk a little bit more about so what is the case for Docker, especially as a developer, why would I care, one would be accelerate developer onboarding and just consistency across developers. You know, how many times have you said it works on my machine. With Docker, it truly does. So if you had a new hire, instead of potentially spending days to get everything installed, with Docker, I can give you a couple commands, and you can get an entire environment, APIs, front end, back end, everything, up and running in literally a matter of minutes in most cases. Another one, eliminate app conflicts. If you've ever had to run a different version of your app and even different version of the framework you use at the same time on a server, sometimes that can be problematic. We might have v1 for some older customers and v2 for new customers for whatever reason. Well, with Docker, that's just two containers. Likewise, environment consistency, and I kind of gave this away a little bit earlier. If you've ever had that problem where you move something to dev and then maybe staging or QA and then ultimately production, and there was just slight differences on the servers between those environments, you know how it's very frustrating. With Docker, we can get those containers running in all of our environments, on anyone's machine, on any cloud, and they will run exactly the same. And ultimately, that leads us to we ship software faster. I mean that's what this is all about. Having gone through the case for Docker, let's review really quick some of the things you probably covered in the earlier course in this learning path. But just in case you didn't, I'm going to walk you through images and containers. So an image is kind of like the recipe for the cake that we want to create. An image is going to have everything we need to run that front-end app defined in something called a Dockerfile, and then we can take that and build it into an image. So think of an image as kind of a bunch of stuff stacked up like books almost. We call it a layered file system. And some of those books are your code. Some of those books are the server code, environment variables, service patches, security tweaks, whatever it may be, everything you need for that front-end or API is in that image. Now if I move that image to your machine, cloud, or wherever you move it, I can then take that and run it as a container because it has everything we need to now get this server and your code up and running, and I'm going to be showing the different commands as we move along to build the image and run the container. Now if we break it down then, an image is really just a read-only template. It has everything we need to run the front end, the API, the database, whatever it may be that your containerizing. And it has this layered file system, again kind of like a stack of books, where every book is a different part of the story that you need to run that particular part of the app. Once an image is moved to a machine and we can actually pull an image from various areas, then we can use that image to run a container, and the container is actually the runtime version. So the container itself is what is started and stopped, and you can pause them and inspect the logs, and this is the runtime version of what we're going to be working with. So now that we've gone through and reviewed kind of the old way to deploy, the new way, a few different cases for using Docker and then images and containers, let's go ahead and jump forward here to the software installation to get Docker running on your machine.

Software Installation

Let's take a quick look at the software you'd need to get Docker running on your machine. Throughout this course, I'm going to be using something called Docker Desktop. Now this will run natively on Windows or Mac. You will need Windows 10 or higher, though, and a newer version of the macOS software. But assuming you have that, you can run Docker Desktop. Now this is going to provide the image and container tools that we need to build custom images and run containers and stop them and those types of things. What's nice about it is once it's installed you can pretty much do everything you need with Docker. Now some of you may be on Linux and going, so am I left out in the dark? And the answer is no because Docker is actually a Linux technology. There's a little magic that's going to go on with Windows or Mac. Windows can use Hyper-V with WSL, Windows Subsystem for Linux, or just Hyper-V alone. Mac uses Hyper Kit. But Linux can actually run it natively because it's based on Linux technologies. So if you are on Linux, although we're not going to focus on that in this course, I will show you where you can go to get the Docker Engine installed. The website that you can go to first is docker.com/get-started, as you'll see here in the lower-right corner. So let me jump on over on my Mac first, show you a few things there, and then I'll show you on Windows as well. So on Mac, when you go to this URL, you'll see Download for Mac right here. And once you've downloaded the file, you can simply run through the installation process like normal, and you will see this little Docker whale up in your menu bar. So if I click on this, you'll notice it says I have Docker Desktop running. I can get to Documentation, go to the Quick Start Guide, something called Docker Hub, which we'll talk about a little bit later. I can check for updates. But I want to show you Preferences real quick. Now the preferences are where you can tweak a few things. So for example, do I want Docker Desktop to start right when I log in? Now I do because I use it a lot, but maybe you don't want your resources taken up on your machine, and you just want to manually start it. Then you could uncheck that of course. I can also come in to Resources, Docker Engine, Experimental Features, and there's various settings there. Normally you don't need to mess with these, but feel free to check those out if you'd like. And then Kubernetes is outside the scope of this course, but think of it as a way to run many containers in an engine that's designed for that. And if you're interested, you can check out my Kubernetes for Developers course. It's called Kubernetes for Developers: Core Concepts. And then there's some other courses after that, if you'd like. So that's what it looks like on Mac. Now, if I go into the Windows side, you'll notice I can download for Windows. I'd get that executable, and I can get the same Docker whale icon. So if I go to our taskbar here, I'll go up to my tray, and you'll notice the Docker icon. If I right-click on it, I can get to the same type of settings. So notice I can get to Settings, not Preferences in this case. Now, I'm using something called Windows Subsystem for Linux. So, yours might look slightly different than mine on Windows. It really depends on how you decide to set yours up. But you'll notice I have the Start Docker Desktop when I log in. And then I'm using this WSL 2 engine. So, I'm actually running Linux, Linux Subsystem it's called. And Docker is running inside of that. You'll notice there's a Learn more if you'd like to learn more about that. And just like on Mac, you'll see Resources, Docker Engine, and Experimental Features, and then there's also Kubernetes. Now what's also nice about this is I could come on in, and just to get started, if I go back to the tray and I right-click, I can go to Dashboard. And this works on Mac or Windows. And notice the dashboard says no containers are running. However, I can copy this command right here, and if you'd like to see if Docker is running properly you can just open up a command prompt. And then I'm just going to paste this command into, in this case it's PowerShell. We're going to do a docker run. Now we haven't covered this command yet, but we're going to be talking about this. You'll see a few command line switches, you'll see this is the image we're going to run, and I'm just going to hit Enter. Now what this will do is it's going to look locally and say, hey do you have the recipe, the image? And I don't, so notice it's downloading it. It's pulling it. And then it just tried to start this container. So it downloaded everything it needed, and now it's going to try to run it on my machine. Now if I run a command, which is docker ps -a, this is list all my containers. We'll be talking more about that in a later module, but we'll jump ahead real quick here. You'll notice I actually have a running container here. It's been up for 19 seconds. I can actually go to the browser and hit it on localhost if I'd like. So let me come on up here real quick. We'll just go to localhost, and there we go. It's now actually running against that container that's on my machine. And notice I have an entire application I can go through here. I can click on just like a normal website. Once I'm done, I can come back to the command prompt, and I can go ahead and say docker stop. And then notice this ID over here. So I'm going to type 10, just the first part of it, and stop it. And then I'm going to go ahead and say, hey I'm done with this. I'm never probably going to use it again, so we'll remove the first part of the ID. And now if I run docker ps -a nothing is there. So this is kind of our Hello World to make sure that Docker is running properly on your system. And this command I just ran is going to run on Linux, Windows, or Mac. Now, let me go back over to the Get Started page. And I mentioned that if you're on Linux you can run Docker Engine. So, if you click here, you can go to View Linux Engine. And this will walk you through what you'll need to run that. There's a couple options you'll notice, Fedora, CentOS, and more. So, that is something I won't be going into here, but you can absolutely run these same types of commands on Linux, Mac, or Windows. So now that we've covered the software installation, let's move forward and take a look at the app that we're going to be covering throughout this course.

Examine the Application

Before we dive in to creating custom images and running containers, let's examine the application that we're going to use throughout this course so you have a little better feel for what's in it and how it works. The application that you see here is built with Node.js and MongoDB. It's a very simple app, all it does is display some Docker commands and what you can do with them, but it should provide a really nice starting point. And I want to emphasize again that even if you do nothing with Node or Mongo or the technologies here, the concepts we're going to learn are going to apply to every other framework out there. And that's what's really important about this particular course. Now to start things off, just to show you where things live, you'll notice server.js. This is kind of our web server, the code that's going to run in it. It uses Express.js. And then we have in our lib folder some code for the database interaction with MongoDB. And this is going to use an npm package called Mongoose to actually connect to that. Now knowing the code isn't going to be required though. Really, we're going to focus just on the steps from a Docker perspective that you need to know to build custom images, run custom containers, talk between containers, and things like that. So to wrap this up, what I'm going to do is run through these steps really quick, starting with the docker-compose build. Now this is an orchestration tool that we'll talk about towards the end of the course, but learning the commands upfront or just at least seeing them is good for repetition. So I'm going to run you through the basics. So let me go ahead and build. Now I'm going to run docker-compose up. And what this is going to do is start up two containers. You'll see we have a Node app and a MongoDB app running now. Now this particular instance of the run shows all the logs, you'll notice, for MongoDB and for the Node app. You'll see the DB connection is open, for example. You can also run it in a mode where it just goes away and you get back the console. And we'll talk about how to do that a little later, but we're going to leave that for now. Now, I'm going to run over to the browser, and I'm going to run to localhost:3000, which is what this is listening on for the actual port. You'll see I have Docker Commands, but there's nothing listed, and that's because the database is empty up to this point. So let's go on back. And what I'm going to do is come on in and start up a new terminal. And once this terminal fires up, we're going to run the next command. We're going to do docker ps -a, and it's going to list our containers. Notice we have a Node container and a Mongo container. Now, I'm going to note the ID here of the Node container. So, I'm going to go ahead and copy that. And we're going to run the next step 6 here. And this is going to shell into the container with that particular ID. And we're going to shell in using an sh type shell. All right, so now I'm actually in the container. And now I'm going to run a little dbSeeder script, which is just a fake, quick way to get some data in the database. All right, there we go, we're initialized. So, I'm going to do Ctrl+C to stop that. I'm going to type exit. And then we're back to our command prompt here. Now let's go back to the browser. And there we go. Now you can see some Docker commands are listed, and it's actually calling MongoDB. The point, though, is we actually have two containers that we're going to be talking about as we go through the course. To finish this up, let's go ahead and run docker-compose down to stop our containers. So now that we've examined the app a little bit, let's go ahead and do a final summary, and then we'll start jumping into all the fun.

Summary

In this first module on setting up your development environment, we talked about the case for Docker, some of the benefits it can provide, and how we can get it up and running on our system. So as a quick review, Docker has a lot of different benefits we can take advantage of from a development workflow or deployment workflow standpoint. Maybe you're onboarding new employees and you need them getting up and running with this project very quickly. They need to be able to run the app locally on their laptop or other machine. Well Docker can make that much easier because you can run the different parts of the app in containers on their machine. We also get environment consistency. It's so nice to be able to run something on your machine and then move it between your different environments at work and have things run consistently and avoid some of those surprises where a different security setting was tweaked on a server or a service pack wasn't available. And then, finally, by using Docker we can ship software faster as well. We saw that Docker Desktop can run on Windows or Mac, but if you're on Linux you can run the Docker Engine. And because Docker is a native Linux technology, it runs great. In fact, that's what Mac and Windows use by default is actually Linux under the covers. It is important to note that Windows does have a Windows container option as well. And the reason I mention that is because Docker can work with many frameworks and languages. Although this course is going to focus on a Node.js and Mongo DB type scenario, if you're doing .NET or Java or Python or PHP or something else you're going to be able to run it either on Linux directly, depending on your framework of course, but if it only runs in Windows, Windows Server for example, then Windows containers might be an option. So now that we've talked about setting up your development environment and some of the benefits of Docker, let's jump into what's next. And we're going to talk about how you can build custom images for applications.

Create an Application Image

Module Overview

Now it's time to jump into the fun, and I do think learning all this is a lot of fun, because once you learn the fundamentals of creating an image and then using that to run a container, it really opens up a lot of possibilities. So in this module we're going to start off by talking about something called a Dockerfile, and we're going to learn about some key instructions and core concepts you can use to define what I like to call the recipe for all the ingredients that are needed to ultimately run a container. Then we're going to create a custom Dockerfile, so you can see that process, and although our Docker file will be related to Node.js, the concepts we're going to go through could be used with Java, .NET, and many other libraries and frameworks out there. Once we have a Dockerfile available, we need to convert it into a Docker image, and we're going to learn about some Docker commands we can use to make that happen. We'll also learn about something called tagging images along the way. Now once you have an image, that's great. You can run the container locally. But how do we actually get it out there where somebody could pull it down to a cloud server or use it on their machine, whatever it may be? Well, we'll learn about something called Docker Hub. This is a registry. There's many of these out there, AWS, Azure, Google Cloud, have these registries where you can actually store your images. We're going to talk about a free option that's part of Docker and explain how you can get your image up into this Docker Hub. Then we're going to wrap up with a really nice VS Code extension. It's related to Docker, and it's something that'll simplify some of the tasks that you would normally do through the command line. Now, I'm going to focus mostly on what you can do from the command line, because I do think it's important to learn the commands first and then learn about other tools that might simplify your life a little bit and maybe make you more productive in some scenarios. So let's get things started by talking about Dockerfiles.

Understand Dockerfiles

In the previous module, we did a quick test to see if Docker Desktop was running properly by running a getting started Docker image, and somebody from Docker would've had to create a Dockerfile in order for that image to be created as well. So in this section, we're going to talk about what is a Dockerfile, how does it work, and what's in it? So before we jump into Dockerfiles, let's go through a scenario that most of us as developers are going to be pretty familiar with and comfortable with, and that is taking code, running it through a compiler, and then generating a binary. So if you've worked with C++ or C# or Java, or even if you haven't, you've probably heard of a compiler and heard about this process where we write some text, that's our code, we run it through the compiler, and we get a binary. Well, that's much how Dockerfiles work, actually. Here's the definition of what a Dockerfile is. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. So unlike writing, say, C++ or Java code, we're actually going to write commands, we officially refer to them as instructions, but they don't look anything like a programming language. They're just text. They're very easy to understand and very easy to work with, you'll see. So here's what it looks like if we convert from our compiling code scenario to building Dockerfile scenario. We're going to write a Dockerfile that has instructions in it. We'll talk about some of those instructions momentarily. We're then going to run that through a Docker build process, and that's going to generate an image. Now that image is going to have everything we need, again, environment variables, the server, security settings, your code. But in order to make that image, we have to make a Dockerfile, and we have to define these instructions. You can think of a Dockerfile as a layered cake. It's officially a layered file system, but if you think of a chocolate layered cake where each layer is a different instruction, that's how Dockerfiles work. The base layer, and the first thing you always define, is the FROM instruction, FROM, in this case, node:alpine, or FROM an ASP.NET Core, or Java, or Python image, whatever it is you work with, that could go here. Now you can go to hub.docker.com to look up the different options. There's a lot of them out there. But in this case, I'm going to say node:alpine, or I could just use Node. And I could even put the version of Node, the version of Alpine, that I want to use, and for production that would be recommended, but we're going to keep it simple here. Now the next thing you'll often see is metadata about the Dockerfile, such as a label for who is the author. Author is the key, in this case, Dan Wahlin is the value, LABEL is the instruction. We can also have another layer, a label, that has our creation date or last updated date or those types of things as well. You can also define environment variables using the ENV instruction. In this case, NODE_ENV=production. That now becomes part of the image, and once we run the container, that environment variable will be part of it. Next up is the working directory. When you're running a base image like Node or ASP.NET Core or Java, there's going to be a lot of directories, of course, in the container once it's running. Well, what's the directory that you care about? What's the directory where, for example, your web server code lives that starts things off? In this case, we're going to say /var/www, and in custom scenarios, you can kind of make up whatever directory you want here. But this is kind of the starting point of "you are here." You are in the /var/www once that container starts running. Now the next command is really important because a lot of times you need to copy in configuration files or settings or code. Could be binaries, compiled code, or it could be a scripted language like JavaScript, whatever maybe you're working with. The COPY instruction has a where am I starting from, that's the dot on the left. Now that would mean it's going to copy from the folder where this Dockerfile lives and anything else in that folder is going to get copied into what's on the right. So what's that dot on the right represent? Well, remember the working directory /var/www. I could have typed that right there. It would have been fine. But we save duplication here. So the one on the left is the source we want to copy from, and the one on the right is where are we copying to, in this case, /var/www. We can also run different commands. In this case, it's a Node app, so we need to have our npm packages installed as part of the image so that it runs consistently everywhere, so we're doing an npm install. Anything you want to run as a command, you could do that here, and you could use the run instruction. I'm going to expose port 3000. That's going to be what this container will be listening on, is 3000. That could be whatever your web server, in this case, is going to be listening on. And then finally we have the ENTRYPOINT. What's the first command that we're going to run to start this up? In this case, we're going to run node, server.js. But if you're using Java or .NET, you'd use the appropriate command-line tool to fire up your executable, if you will, your starting point. Now, where do you get information on all this, because now we're looking at quite a few instructions, and you may say, well, hey, I want to know more about the label or the environment variables or something like that. Well, fortunately, in the Docker documentation site, you can get information on all these commands. This link here will take you right to that. And then from there you can drill into every instruction and get more information about it. But this is a quick overview to get you thinking about what is a

Dockerfile. So now what we're going to do is switch gears to the app, and we're going to look at a custom Dockerfile for that app. And then a little bit later, we'll talk about how we can build it into an image.

Create a Custom Application Dockerfile

All right, well, now that we've talked about Dockerfiles, let's look at how we can create our own custom application Dockerfile. So coming into the application, you'll notice I have a file in here called node.dockerfile. Now you could just name this dockerfile with no extension, but I generally prefer to be a little more specific, because in many applications, I'll have multiple Dockerfiles. Now keep in mind, I have this at the root. If you wanted to move it to a different folder, you could do that as well. There would just be some relative paths that you may have to change, and I'll show you that coming up. So what I'm going to do is open up this file and I'm going to delete everything that's in here to start us off. Now the first thing we want to do is give it the base recipe, I'm going to call it, the base image, and I'm going to use node:alpine, in this case. If I run on over to a site called hub.docker.com, which you'll see here, and search for the Node image, which I've already done, then you can find more information about the version of the image that's available. Now I did node:alpine, so if we scroll on down, you'll notice 15.9.0 looks to be the latest version, so we could be very explicit and say 15.9.0-alpine or just 15.9-alpine, or we could even do current-alpine. If you do what I'm doing here, it'll grab the most current version, which on the surface probably sounds like a great idea because you're always up to date. Actually, it's not a great idea, and the reason is that as you build into the future, there may be changes to this image which could affect your application. So be really, really careful that you're more explicit. Now I'm going to go ahead and leave it, but we could certainly come in and do 15.9.0-alpine like that, and that would actually allow me to be very explicit about the base image that I want to base mine on. I normally do that, especially when I'm not just doing a demo or playing around. If it's for production, you would probably want to do that, so keep that in mind. Now I'm going to go ahead and leave it as current, though, for this demo. Now the next thing I'm going to add into here is optional, but I would highly recommend it, and it's a label. So I'm going to put LABEL, and I'm going to hit Tab here. Now this one is going to be author, and then you can put whoever the author of this initial Dockerfile. A couple of ways to think of this, one, you know who to blame if something goes wrong, not a good reason. Number two, though, and more seriously is, at work, if you open a Dockerfile, you can find out maybe who on your team initially created it. But this is a nice way to have it right in the Dockerfile. Now, you can put really any key/value pair you want here. We could add another label for the date this was created, for example, or something like that. Now, the next thing I'm going to add is instructions for environment variables. Just about every framework out there has some type of environment variable that you can set for, is this development or production, or you might define a version of the app or framework, things like that. So I'm going to go ahead and just paste these in to save on typing. And we're going to say that this is going to be our production version of our Node image, so NODE_ENV=production, and I'm going to define the port here through using an environment variable. All the ingredients I need, including environment variables, can be supplied in this image. Now coming on down, we're going to work with a directory, and this directory is going to be /var/www. Now, this is something I came up with, it's pretty common, actually, but this is the directory inside of the running container. In this case, it's the directory in the Linux container that'll run this node:alpine image. So what I'm going to do is say this is my working folder so that moving forward, we don't have to keep typing the path to everything we want to do. We can just type it once. Now when Docker does its thing, it knows that that is the working directory, and then it figures everything out for us. Now, the next thing I'm going to do is copy in some files. You'll notice that on the left here we have our package.json and our package-lock.json. Now not only do I need those in the image, but I need them because we need to run npm install. So I'm going to go ahead and do a run command here, and we'll go ahead and just delete that, tab on over, and I'm going to do npm install, and you might even have other flags you want to add to that such as no optional dependencies, for example. You may wonder why we're copying in just the package.json and the package-lock.json and then doing the npm install. And the answer is because Docker works with layers, and this keeps the layer focused just on the package.json files. And now if we have other source code changes, that would update a separate layer if we rebuild this, but it won't affect the npm install part. Now, I am going to need to copy all the source code in, though, of course. So I'm going to do a COPY command here, and we're going to copy in from the current folder to our working directory. Now, I could just do that, but to be really explicit, let me go ahead and show you, it could be that as well. Now, that would be kind of redundant, though. So a lot of times you'll just see something like this because we already said /var/www is my working directory. This first one says copy this code from my local file system. The second one then says into /var/www. That's what we'll be doing there. We have a port up here. I'm going to go ahead and say that this image and the container, ultimately, is going to expose a port. But I don't want to hardcode 3000 there because, who knows, maybe I want to use this in multiple places. In this case, it's just once. So the way we can get to an environment variable is like this. We could put a dollar sign and then put the key that we have for our environment variable. So if I wanted to get to NODE_ENV, down here we could put a dollar sign in front of it, and then it would substitute it as it does the Docker build. Finally, I want to say what to run when this starts up, our entrypoint. So I'm going to go ahead and say that there's an npm start command. Of course, that's going to be in our package.json file. Or if you're using other package-type managers for your framework, you could have that initial command here. Something really, really important I want to point out. In this kind of demo app, it's not as big of a deal, but in larger apps, it's a big deal. A lot of times, once you do a build, and we'll look at that coming up shortly, you'll find that it takes forever sometimes to start the build. And you'll sit there and look at the console and go what is going on? And when we do a command like this, it copies everything in, which, in this case, I want just about everything, but I definitely don't want my local node modules. We're going to do that right here. So how do we ignore those so they don't get copied in? Because that could take a long time, in some cases. Well, notice at the root I have this .dockerignore, which has Node modules. That will stop Node modules from being copied in. It's much like a gitignore that you'll see here, which has some other things, like Node modules. So keep that in mind that files you don't want, just add a dockerignore. And then when you do the build, you, first off, won't be slowed down. But second off, your images will be smaller because you won't be copying unnecessary things. So that's a look at how we can create a custom application Dockerfile. So the next thing we're going to do is take this and build it.

Using docker build

Once you've added all your instructions to your Dockerfile, you need to run it through the docker build process, and fortunately doing that's very straightforward. So in this section, we're going to take a look at what the docker build command looks like and how you can use it. So to get started, the simplest way to build an image would be the following. You'll type docker build -t, that stands for tag, or you can do --tag if you prefer, and then you can give it the image name. And the image name could be very obvious, like customersapp, nodeapp, javaapp, whatever you want to call it, normally you'll be very, very specific on that name. And then the final thing you'll do is give it the build context. Where is the Dockerfile relative to where you're running this command? In this example, I'm saying that it's in the same exact directory where I'm running this command, that's the dot. If it was at a higher level, I might put ../ some parent folder. If it was a lower level, I might put ./ some sub folder. So that's something you can control, but normally you just go right where the Dockerfile is, and then you can run the build command, and just put a dot. Now here's an example of putting that into action then. We have docker build -t, and the name of the image in this example is nodeapp, and then there's our build context. While this type of approach works fine, at some point you're going to want to push your image up to a registry. That's a place where you can store images. Now this could be a registry that set up within your company, it could be Docker Hub, that's the hub.docker.com we've seen before, it could be up on AWS, or Azure, or Google Cloud, or some other cloud. That depends on you, your company, and where you're storing your Docker images. So when you do that, normally what'll happen is you'll go set up a registry username, you'll then use that username as part of your tag process. So let's walk through this here. We're still going to say docker build -t, that stays the same, but now we're going to put our Docker registry name. For me, up in Docker Hub, it's danwahlin, that's how I registered it. But again, yours could be your company name, your department name, your team name, it really depends on who set up the registry. So if you go to hub.docker.com, and you register as yourself, then you'd probably pick, you know, your name, your first/last name, maybe, let's say, and that's what would go here, and I'll show you an example of that in just a moment. Now you put a slash after that, and then you put the image name, like we already talked about. Now there's one more really important piece though, and this is the actual tag part of the tag. You can do tag here, and you're going to see in a moment this is normally the version of the image. Why is that important? Well, we just saw earlier that with node and other images out there, they're always updating the images, and they tag them with a specific version. That way, when you pull down an image, you can pull down a very specific version, not just the most current version. So, for example, I might want to do the following then, I might want to docker build -t, danwahlin, in my case, is my Docker Hub registry name, nodeapp would be the image name, and then I'm putting a colon 1.0, 1.0 represents the version. Now let's say we do some changes to the code, and now we want to update this. We do another docker build -t danwahlin/nodeapp:1., let's say, 1, or 2.0, or whatever it is you do for your versioning at work. So you can see that running the docker build command is very straightforward. There are some other command line switches, but this is really all you need to know to get started, and we're going to use this in just a moment. Now before we do, and we actually build our Dockerfile from the app, let me show you a few other commands I'll be demo-ing here. You might have seen these before, but docker images is important because sometimes we just want to list what Docker images do we have available, and then docker remove image, rmi, and then you give it an imageId afterwards, and that's how you can basically remove or delete an image. So now that we've seen some of these different commands, especially the docker build command, let's jump over to the app real quick, and we'll go ahead and build that Dockerfile.

Build a Custom Application Image

So here's the Dockerfile that was created earlier. Now to build this, we could do docker build -t, give it a name, I'll just say nodeapp to keep it simple, and hit dot, you would think, but notice what happens here. I get an error, and the error is because it can't find the Dockerfile. Now you're going to look over to our files and go, it's right there, why is it missing it? Well, if I were to rename this to just dockerfile, a lot of times you'll see it like this, then this would work actually. In fact, let me go ahead and do that, and you can see it's actually doing the build right now, and it's done. So if we clear this out and say docker images, I have quite a few in here, but there is nodeapp right there. I didn't tag it with a version, so it's marked as Latest, and then here's the IMAGE ID. Now c34, let's remember that because I'm going to go ahead and remove it, c34, and there we go. Now it's deleted, and if we did docker images again, then it would be gone. Now that's great, but that's not how I named mine, right? I named mine node.dockerfile. Well, fortunately, it's pretty easy to get some help. We could run off to the docs, or we could just do this, we could say docker build --help, and you're going to see there's a -f switch we can use, file, Name of the Dockerfile. So the default is whatever path you're on, Dockerfile, as you just saw, but in our case, we're going to have to be a little more explicit about where the Dockerfile is as far as the name. So we're going to do docker build -t. Now in this case, instead of just saying nodeapp and then hitting a dot, I'm going to do -f, and I'm going to say node.dockerfile like this. Now let's hit Enter, and now notice it says it requires exactly one argument. What am I missing here? The dot. This is the most easy thing to miss. Nowadays, I'm pretty used to it, but it used to be when I was new, I would stare at that error and go, why isn't this working? Well, I'm just going to add a dot because we're in the folder where everything is located, where node.dockerfile is located. So I'm going to hit that, do Enter, there we go. Now this built really, really fast because it was already cached. In fact, that shows off some of the cached layers you'll see right there, but if we do docker images, scroll to the top here, there we go, there's our nodeapp right there. Now let's delete it again, 837 is the IMAGE ID. And let's finish this off by doing a little more formal way of defining this. So I'm going to go back, and this time I'm going to put my Docker Hub username. So I'm going to put danwahlin, that's my registry name, nodeapp, and let's go ahead and say this is version 1.0. All right, let's hit Enter, super fast again, and now watch what happens though when I do docker images. Scroll up, there we go, danwahlin/nodeapp 1.0. I could rebuild again, let's do that. Let's say we changed a file or two, and we could do 2.0. All right, that's done, and now let's do docker images, and now we have two of them. Now these happen to be identical, of course, because I didn't change anything. You'll notice the size over here. But that's how easy it is to tag. So the question comes up a lot, well, how do I update my image? The answer is you don't. Images are immutable. You make a new image, and you tag it. So that's an example of using the docker build command in a few different ways. You saw the - f switch, but you also saw that the dot really matters, the name of the file can matter, and the tags really matter, especially as you're building images out into the future. You want to version those.

Deploy an Application Image to a Registry

Once you've created and built an image, you need to deploy it to a registry. Now a registry could be local within your company, but oftentimes they'll be up in the cloud somewhere, such as Docker Hub, which is what we're going to look at here. So the process of pushing an image up to a registry such as Docker Hub, or really any other, is very, very simple. You could do docker push, and then give it your username, if that's what you're working with, followed by the image and the tag, like we saw earlier. Now that's all we would have to do. For Docker Hub, this would be your Docker Hub username, of course, your image name, and then the tag that we just talked about. Now what that will do is push that up to Docker Hub by default, it'll be publicly available, but there's even behavior there that you can change up in Docker Hub if you want to make it so it's private to just you, so that nobody else can get to it. So let's go ahead and jump into an example of actually using the command to push the image that we have up to Docker Hub. Here's the dockerfile that we looked at earlier. I'm going to go ahead and do one more build, so we know it's kind of a fresh copy of the image. Notice I have my docker build -t. Danwahlin is my Docker Hub registry name, nodeapp is my image name, and I'm going to tag it with version 1.0. And then because my dockerfile is named differently, it's not just named dockerfile, it's node.dockerfile, you'll notice I'm using the -f switch, and then I'm giving the context of where is this dockerfile. And we're all done. So now if we go back to docker images, and there we go, danwahlin/nodeapp 1.0, and you'll notice the image version that was just created. So now what I want to do is get this up to Docker Hub. The first thing you'll have to do is a docker login. Now, I've already logged in, so I'm not going to do that process here, but as a heads-up, if you have two-factor authentication enabled, you'll have to create an access token, and that's something you can do if you go to hub.docker.com. Now I've already done that because I do have two-factor enabled. If you just have username and password, you can log in like normal. Now from there we can go ahead and say, I want to push this to danwahlin. Now because there's no other domain in front of that, there's no IP address, there's no domain, it's just danwahlin, it defaults to Docker Hub. Now that's overridable, but that's what you're going to get out of the box with Docker Desktop. So we'll go ahead and put that, and then we have nodeapp:1.0, and that's all we need to do. So I'm going to go ahead and push this, and notice it's pushing every single layer that's part of this image up to Docker Hub, and then we'll go ahead and look at Docker Hub in just a moment. Alright, so let's jump over to the browser real quick, and we'll take a look at what's this look like once it gets up to Docker Hub. And here we go. Notice I have my danwahlin account, and there's my danwahlin / nodeapp. We can go ahead and click on this, and we can get some information. It looks like I only have one version up here, it's for Linux, and it was just uploaded or pushed. And then here's how we could actually do the push if we want to do it again. So if I want to update with a new tag name, we could do another docker build, give it a tag of, say, 1.1, and then I would have two of these up there. So, as I mentioned earlier, images are always immutable. You're never changing the image, you're always moving forward, and you'll just simply tag it with a different version. So that's all we have to do to push an image. Now, of course, from here we can go on back. Now if I wanted to pull this to another machine, I would do docker pull, like we saw earlier, danwahlin, and then we give nodeapp 1.0. And then that would pull it to my machine, I already have it, so I don't need to do it. Now from here, we could go ahead and run it. That's actually going to be coming up in the next module. So now that we've looked at the push process, I'm going to show you a really nice extension for VS Code that can save you some time with all of this.

Use the VS Code Docker Extension

Up to this point, we've been using the Docker command line tools to build our images, push them, pull them, things along those lines. I personally feel you need to learn the CLI commands first, the command line interface commands, but once you know those, there's nothing wrong with using other tools that might increase your productivity, of course. One of those is the VS Code Docker extension, and I'm going to show you a quick look at a few of the things it can do. So, as mentioned in this module, we focus exclusively on the CLI. We've jumped down to the command prompt and run docker build, docker push, docker pull, and other commands. There is another way, and that is the VS Code Docker extension, and it's a really nice extension. You might have noticed that first off, I've got some nice code syntax highlighting here going on in my dockerfiles. If I hit Tab on a few things, so for example, if I come on in and do FROM, notice that I get some help over here and I can get to the online docs. If I hit Tab, I can do that. And then if I type my image, if I do node:alpine, it will actually go out and show me the different versions that are available. Now you might notice over here I have a little Docker whale, and if I click on that, I could get to my images, such as, there's my nodeapp, there's my 1.0 that we did. I can get to any running containers, registries, so I can actually connect two different registries, Azure, Docker Hub, some generic one at work, GitLab, and you can add others. And then I can even go down and get other information. Now, to get started with this, you can come into your VS Code extensions, and if you just type docker, and you'll see this docker extension here. And this is actually published by Microsoft. You'll notice it's downloaded a lot, gets high reviews, and this extension does all kinds of things. As I mentioned, it will help you out with your Dockerfiles, but it also has a lot of built-in commands. And I'll let you look through this if you'd like more details, but I'm going to show you a few of these in action. So if you install this extension, you're going to get some nice options, and let me show you a few of those. First off, I can right-click on any dockerfile, and towards the bottom you'll notice I can Build Image right to Azure, Azure Container Registry, or I could just say Build Image. Now let me go ahead and do that. Notice it pops up what it thinks I might want. Well, I don't want that. I'm going to do danwhalin/nodeapp, and let's say 2.0, and now I'll hit Enter. Now watch in the command problem down here. Ah, look at that, it's actually making sure that we're pulling everything we need, and then it's doing a build, and there's our tagging that we've already seen up to this point, it's just doing it for me. Now, we'll let this go ahead and build. It looks like it's all done, so it says Press any key to close. We'll go ahead and do that. Now I can click on the Docker whale, come on up, and we should have 2.0, there we go. So you don't even have to type docker build if you don't want, you can let this do the build for you. I do that actually quite often to be honest. I know the command well, but especially in cases where your file is not just called dockerfile, you know, like mine is node.dockerfile. It's a lot easier to just right-click and say Build Image, and so I'll often go that route. There's a lot of other things you can do with this, but for our build module, those are the main things you can do. So to give you some syntax highlighting, help you with your image versions, help you with your docker commands as you're typing, plus, you can right-click on your dockerfiles and go in and do things like Build Image. Very nice. Now, there's a lot more than that, though. I'm going to show you one more, because it segues nicely into what we're going to cover next. If I go back to here, let's say we want to come to the 2.0, I can right-click and look at that. I can run it, I can tag it even, so if I want to come in and tag this with a different version, 3.0 or something, I could do that very easily. And I can do other things such as inspect it, and this will actually show me a JSON-type data file of all the information about that particular image. And then I can even push and pull right here. So we just did a push in the previous section. I could go ahead and do a push now. It gets it ready, there it is, I can hit Enter. And there we go. Now it's pushing up my 2.0 image up to Docker Hub for me and I didn't have to type anything. So that's an introductory look at the VS Code Docker extension. We focused on building images in this particular module, and you've seen that and how to push them, but it can do other things as well. Very nice to know about.

Summary

We've focused pretty exclusively on creating a Dockerfile, building the Dockerfile, and then pushing that image up to a registry, in this particular module. I hope that gives you a really good idea about that process and some of the commands. As a quick review, we talked about how Dockerfiles provide instructions, and they describe the layers that ultimately are going to be part of the image that you create. They can have simple things like metadata, they could have tasks that should be run, define ports, and there's a lot of other instructions that are available. Once we have a Dockerfile, we can run the docker build command. That can be used along with the name of the image, and oftentimes, in more real-life scenarios, you'll define your registry name, whether it's your username, your department, your company account, whatever that may be, along with the image and the tag. Tagging, I mentioned, is a very important aspect of this, because while you could just make every image you build the latest, latest can get you in trouble because there might be some new features that your app just wasn't planning for in that image. So by always tagging an image with a version, you'll always know what you're running exactly, and it's a lot safer way to go, especially when you get a lot of images out there and a lot of applications and containers out there. When it comes to pushing your images, Docker Hub provides free storage that you can use, but there are other registries out there. You might use Azure, AWS, Google Cloud, your company may even use a product where you push an image to that particular registry. And then the VS Code Docker extension really simplifies several tasks, and although, I mentioned earlier, I highly recommend you learn the Docker commands first and run them from the command line, once you know them, as I said, there's nothing wrong with being productive, in fact, I'm all for it, and this can help you be really productive when it comes to building, tagging, running, pushing, pulling, all those types of things, so it's nice to know about, and something that's very easy to get installed in VS Code. So now that we've talked about Dockerfiles and the build process for images, it is time to move on to how do we run these images as containers, and that's what we're going to look at in the next module.

Run an Application Container

Module Overview

You've seen how to build a custom image, so now it's time to move on and learn how to run that image as a container, really anywhere you'd like to run it. It could be on your machine, could be on a server at work, could be up in the cloud. So let's take a look at how we can get started here and what we're going to cover. First, we're going to review how we can use docker pull to retrieve a custom image from a registry. If you recall in the previous module, we looked at building a custom image and then pushing that up to a registry. So we're going to pull one down, make sure we have it on our machine. From there, we're going to talk about the docker run command and how you can use it to get a running container going. Very important, obviously, because that's really what this is all about. So we'll talk about different command line switches you can use there, and we're going to talk about once a container is running or even if it's stopped, how can you access log information about that container? Every now and then a container may crash, or maybe it's running and you just want to see the status of something in the logs. There's a command we can use called docker logs that will let us do that, and I'll explain how that works. Finally, we're going to look at how we can store data more persistently using something called volumes. Normally, when a container writes data to a file, it stores that in the container. But what if the container goes down or is just deleted, removed? That would be a problem. So we're going to look at how we can store data outside of a container while that container is running, but yet still be able to access that data in the container while it's up and running as well. Let's go ahead and jump in to a look at how we can get started running an application container.

Run an Application Container

Once you have an image on your machine, you can use a docker run command to get that running as a container. So what we're going to do is review how to get an image and what an image is composed of, and then we'll look at that command in action. So earlier I mentioned that images are really like a layered cake. They're a layered file system, and each layer has a different instruction. It could be your code, could be an environment variable, it could be the base image that you're running, like a server, or whatever it may be. Now each layer then builds on top of each other, and it actually gets an ID. Looks like this. So if we had an image that runs on Ubuntu, that image then would have different layers to it. So each layer gets assigned an ID, as I mentioned, but these are all read-only layers. Once you put these layers in place, they're fixed, they're immutable, and if you want to make a new image, then you build a new image and you tag it differently, as we talked about earlier. So where does a container fit into this? When you do docker run, how does it take that image and make it into a container magically? Well, the answer is, if you think of this as the base of the cake, we need some frosting on top. From a container standpoint, the frosting, if you will, is this then readable/writable layer. So really a container, it is the instructions and the layers of the image, but then it adds this extra readable/writable layer on top. Now the good news is you don't even have to know that, you just have to know the commands, but this is what's really going on behind the scenes when you run a container. So the first thing we'd have to do to run a container is, of course, pull it. We've already seen this. You would say docker pull, give it the name of the image. Now once we have the image on our system, we can then use the docker run command. I'm going to show you a basic example of getting started here, and then we'll jump over to some demos. So, one of the first things you'll have to do is define ports. You will use the -p flag, so docker run -p, and then you define an external port. What's the port people would hit from outside of your container to interact with it? So, for example, if you pull up a browser and type localhost:8080, then the external port would be 8080. Now what's the internal port? Well, remember earlier in Dockerfiles, you can expose a port. That would be the internal port. That's the port that the container and whatever's running in the container is listening on. So external is what external users or systems would call. That then forwards it to the internal port inside of the running container. And then finally, just like with a docker pull you have to put the imageName, you would do the same thing here. You would say docker run. At the very end, you'd put the imageName. So let's take a look at an example of how we can use the docker run command.

Using docker run

The first image we're going to get running as a container is something called Nginx. This is a very high-performance server that can serve up static content and do a lot of other things as well, but it's very easy to get going and a great way to learn about docker run. If we go to hub.docker.com, you can search for nginx, and then if we scroll on down, you'll see the different tags it has. I'm going to pull the nginx alpine version that you'll see down here, mainly because it's a lot smaller and it's very easy to pull that way. So let me jump over to a command prompt, and we'll do the pull, and then we'll run it. So the first thing I would do is type docker pull, and then nginx alpine. Go ahead and hit Enter. This will now pull that image down to my machine. Now, before we run this as a container, let me show you how we can get help on docker run. So if we type docker run --help, then we can get some information about the different commands as we run this. So you can see there's a bunch of commands on this, and the good news is, you only need to know a handful of these. The main one we're going to focus on now is going to be ports, and this will define what we're going to publish to as far as ports go. So we'll talk about that in just a moment, but you'll see there's a lot of command line switches that you can do. I'm going to go ahead and clear that, and let's go ahead now and run docker images, and you'll notice that I now have an nginx alpine. It's about 22MB, and it was created about 2 days ago, so we have a very up-to-date version it looks like. So how do we run this, then? Well, if we go to the Docker Hub nginx docs, you could actually learn how to do it right there, but let me just show you how it would work here. Let me clear this, and we're going to do docker run -p. Now we need to decide what is our external port. If we do localhost for this machine, how are we going to get to that? I'm going to say it's 8080. Now, internally, if you go look at the nginx image, it actually runs on 80. So we would call 8080. It'll forward it inside of the container to 80. The last piece we have to do just to get this up and running very quickly is put the name of the image, so we'll go ahead and do nginx alpine. Now if I hit Enter right here, it'll lock up my console. So I'm going to show you another little command line switch called detached that's nice to know about. So if I add a -d, what'll happen is it'll run the container, but it won't lock up the console and write out all the container logs. Instead, it will just run it, it'll give me the ID of the running container, and then it'll just give me back the console, so now I could do other commands. Whether or not you use this depends, because by putting -d I'm not going to see the logs, but we're going to cover that later. I'll show you how we can get the logs another way. So let's go ahead and start this up, there we go, it gave me back a container ID. Now I can do docker ps -a, and it shows that, yep, I have nginx alpine. There is the status, it's been up 5 seconds, there's the port forwarding, and then it came up with an interesting name because I didn't give it a name. Now we could have also come in and done this. We could have said that the name is my-container, or something like that, and that would have worked as well. So just keep in mind there's a lot of command line switches you can do. In this case, I don't really care about the name, I'm okay with it, but any time you want to look them up, just run that --help. Let me show you the name real quick, so let's go up to the n section here. And there we go, there's --name that I showed you, and it assigns a name to a container if you don't want it to make up a name. How do we verify this is running? Well, let's go to the browser. I'm going to open up another tab. We'll run off to localhost 8080, and there we go, there is our nginx container. Now if I go back to the command prompt, we can go ahead and stop this container. So I can say docker ps, and ps will just list the running containers, okay, mine is running right now, -a lists all of them. So if I want to stop this, I can say docker stop and then give it the first part of the ID, ce4, in this case. Ce would be enough, it looks like it stopped it. Now I want to completely get rid of it. If we go back to docker ps, notice first off it doesn't list it. It's still hanging around, it's just not running. Let's do a -a, there it is, notice it's exited 17 seconds ago. So to remove it, I can do a docker remove, rm, and put a ce. There we go, now let's do docker ps -a, and we have nothing. It's as if it never existed. Now the image is still there. If we go docker images, there's our image, so I haven't deleted that. We could do the docker rmi command, though, to delete the image based on the ID if we wanted. So that's how we can get a container up and running.

View Container Logs

Docker makes it really easy to view the logs for a running container, and this is super useful anytime you're trying to troubleshoot or you just want to find out additional details about what's happening inside of a container. So we're going to take a look at a quick command called docker logs, and then I'm going to show you how to use it with our application container. We'll get that running, and then we're going to view the logs for it. So docker logs is very simple to use. You simply type docker logs and then give it the containerId. Now, to get the containerId again, you can do docker ps or docker ps -a, and that will allow us to then get that ID, pass it the logs, and then it will stream those to our console. So let's take a look at how that works. So in the previous module, I created a danwahlin/nodeapp, and we had two tag versions, 2.0 and 1.0. So what I'm going to do is get that running now. So let's go to VS Code. So the first thing I'll do is make sure I have the latest version of that image, so I'm going to grab the 2.0. This should be cached on my machine already and should be pretty quick. All right, so it looks like the image is up to date. We're good to go there. Now, let's go ahead and run it, but before we do, I want to call out that this image makes a database call, and we don't have that container running, so I do expect some errors here. Now, if you recall, in the dockerfile, we exposed port 3000. You'll see that right up there. So now, we're going to do the same type of thing. We can do docker run -p, and then I could do 3000 on the external or I can change it. I could do, maybe I want to go 8080 to 3000. This is totally up to you, what you'd like to do here. Now let's say we'd like to leave it as 3000 external, and then we know it runs on 3000 internal, and then I'm going to run this as -d, detached, and then we'll go ahead and put the name of it. All right, so let's go ahead and start that up. Now it looks great, right? We got back an ID. Everything's probably fine. Or is it? Let's go ahead and check. So if we do docker ps -a, to show all, including any containers that aren't running, you'll notice it's been up 12 seconds. Okay, well that looks good, so let's see what happened then. Let's jump back to the browser. So I'll go ahead and open a new tab, and we'll go to localhost 3000. All right. And it couldn't go to the page here, so it looks like something's not quite right. Now this is where docker logs can be really useful. So let's go on back to VS Code to the command prompt there and see what we can find out. So I'm going to go ahead and we'll make our console a little bigger. Let's clear it. Let's go back to docker ps -a again. All right, now you'll notice it says exited 36 seconds ago, so something definitely happened that wasn't good. And I'll give you a hint, I've already mentioned it. We don't have a database right now. But if you look at it right now, you're kind of stuck, right? You're going, what do I do? Well, this is where docker logs come in. So if we did docker logs, then I'll give it the ID 58. Let's see what we get here. All right, so if we come on in, oh, there we go. We've got an issue. Looks like there was a Mongoose issue, and Mongoose is a Node.js package that can hit MongoDB. And it looks like it could not connect to MongoDB there, that getaddrinfo mongodb. So we definitely have a problem. And of course, I could scroll through, and it looks like the error tried a couple times to connect, and it wasn't able to connect, so it tried again, and then it failed. Had you not known about docker logs, you'd really be up a creek though. What do you do when you don't know? Now, to show you, we could have done this. We could have come in, let's go ps -a again, and let's remove this, so we'll say remove 58. All right, now it should be gone. Perfect. Now let me go back to where I ran it, and let's take out the -d. Now this would actually stream the logs right to the console here, which you normally don't want because it ties up your console. But in this case, it would have been helpful. So let me just hit Enter here and we'll let it run. All right, now that's what's actually happening behind the scenes. That's the logs. And now if we try to hit it, we're going to get an error, and obviously, it's not going to work. In fact, we should see an error here in just a moment anyway. But let's go ahead, we'll officially try to hit it. Let's go back to the browser, and we'll go ahead and just refresh this. Should get an error again. And at this point, the container might have already crashed. Hard to say, but, yeah, definitely isn't working. So let's go back now to the console. All right. And yeah, you can see the container crashed. Our logs actually took us to the very end, and it exited. That's why down here at the bottom we're no longer seeing logs. Normally, when you're going to run a container then, you're probably not going to want to see the logs. Instead, you can do -d, and that just gives you the ID back, but if something goes wrong, now you know you can use docker logs to get that, even on a stopped container. So, one more time we can now use this one, 86, and there we go. We get the same type of errors again. But docker logs are one of the most useful commands. I use it all the time in Docker. I use it sometimes in even Kubernetes scenarios, which is a way to run multiple Docker containers. So it's a very good command to know about, and that is how you can get to your container's logs, whether it's running or if it's been stopped.

Using Container Volumes

As you work with containers, there's going to be times where you write files or you have database files or others that need to be saved outside of the container. That way, if the container is removed, you don't lose your files. Well, that's what volumes can help us with. So if you had a container that writes out logs, instead of storing those in the container, you could actually use volumes to store it outside of the container. You can store it actually in many places. The default that we're going to talk about is on the Docker host, wherever your containers are running on. So it looks like this. If you have a container that has a var/www/logs folder, and let's say your Node.js app, or .NET Core, or Java app, or Python, whatever it is is writing to that logs, would you want to store the file in the container? Well, you could, but again, if the container dies or gets removed, you would lose that file because everything associated with that container would be gone. Well, in Docker, we could create a volume mount. And it creates this mnt kind of mount type of directory on the Docker host, and now, if your application writes to var/www/logs, as you see on the left there, it would automatically write it to the mount directory location. Now, if your container goes away, gets removed, for example, you're okay. Your files would still be there. And let's assume you're appending two log files. Well, now if a new container comes up that writes to var/www/logs, then it would just append to the existing log files that would be over on the Docker host in the mount directory. Now the way this works is when you do a docker run command that we've already seen, you define your ports and the image to run, you can also use a -v switch that you see here, and this is going to say, hey, Docker, I have a var/www/logs folder. Would you please take anything that's written to that folder and actually write it over to the host? And it will create the directory structure that you would need on the host, by the way, you don't have to worry about that, although I'm going to show you how you can override that if you want. So in this case, any time the app, in our case a Node.js app, writes to that logs directory, it's actually going to be writing to some directory that Docker manages on the host. Now if we take down this Docker container, no big deal. That log file would still be available, and we could get to it. Now what if you want to control the folder location where it writes to? Maybe there's a specific directory you're running the docker run from and you want it to go in that directory. Well, you'll notice a syntax here, pwd, print working directory. Now this is for Mac or Linux. If you're on Windows, it's a slightly different syntax with, for instance, PowerShell, but I'll show you that in a moment. But what this does is the value on the left is where you would like to write to on the host. In this case, if I was in a folder called temp, and that's where I ran docker run from, then pwd would be temp, and it would actually write all those log entries to the temp folder. So you kind of get the idea. Now the one on the right is the container directory or folder. And now, anytime, again, something's written to var/www/logs, it'll actually be written to whatever the pwd is. This provides a really easy way to do a couple things. First off, in production scenarios, it would allow the log files to stick around even if the container's removed. But as you're going to see in a demo coming up in the next section, you can also use this for coding type of scenarios as well. Now what's this look like on the Windows side? Let's say you're using PowerShell. It just changes a little bit. Notice it changes to kind of curly braces instead of parentheses, and you put PWD, like you see here, and that's what you can do if you're running on Windows within PowerShell. So now that we've taken a look at what volumes are for and how they can be used to store data outside of a container, let's look at an example of using volumes.

Create a Container Volume

Let's take a look at container volumes in action. So the first example I'm going to show you goes back to our nginx example that I talked about a little earlier. We can run an nginx server using the docker run, and then we'll give it nginx:alpine. Now as I start this up, I didn't do the detach mode, so you're going to see the logs are going to stream here, which is fine. Now let me run over to the browser. I'll go to localhost:8080, and you can see we have nginx as expected. Let's refresh, make sure it's not cached. Everything's good. What if I want to change the page, though? Well, we could make our own image, and so we could copy in our own index.html file and change that home page. But we can also use volumes to develop live against nginx. So coming on back, let me go ahead and I'll do Ctrl+C here to stop it. We'll clear that, and then I'm going to go ahead and do a docker ps. And notice that it's gone, and that's because it's not in detach mode, and that server shut itself down. Let's go ahead and do -a though, and it's still there. It's just exited. So let's go ahead and remove it, Alright, it's all gone. Now, what I'd like to do is actually have a folder, and I'm just going to make one in here. Let's just call this nginx, and then I'm going to cd the into that folder. And now what I want to do is in the nginx folder, I'm going to add an index.html file. And now in VS Code, I can just type html tab. We'll do body tab, and then let me just do an h1, and we'll just say Hello from my custom page. Now nginx actually has that default home page in the container, so it's part of its image, and you can look up in the docs what that folder path is. Well, what if we use the volume to say, don't go to that path in the container, go to our nginx folder, our pwd, if you will, because I'm currently in that folder here. Very easy to do. We can do the following, so we can come back to our docker run, but what I'm going to do is add a -v. Now I'm on Mac in this case, so I'm going to do pwd like this, colon, and then I'm going to paste in the path that they use. And you'll see it's this /usr/share/nginx/html. Now that is the folder that normally has that default home page that you've already seen for nginx. What I'm now telling it to do is don't go to the containers folder. Instead, alias that or point it to my working directory, which in this case, is going to be my nginx folder. Alright, so let's go ahead and run this now. Now let me run back to the browser. So I'm going to go ahead and refresh, and we should see my magical, phenomenal home page. There we go. Now it's not much to look at, but that's pretty neat. And this is live. I'm actually doing this without even building a custom image. Let me go back. Let's change this to page with, I don't know, a bunch of exclamations. We'll go ahead and save it. Let's go back to the browser, hit refresh, and there we go. Now this works with anything. You could be running a .NET Core server that points to where your C# build is located. You could be running Java. You could be running Python. Node.js is what we're going to look at next, but a different scenario. But that's more of a development scenario for working with volumes. Then we switch over to my Windows machine, and what I'm going to do is run our nodeapp custom image, convert that into a running container, but I want to use a volume. This is going to be for more of a production-type scenario. You'll notice here in the server.js file that I have some code that tries to create this access log stream, and it's going to store in /logs/access.log. That will write out all the browser requests and things like that. Now I'm just going to use a kind of default express-type package called morgan to do that. It doesn't really matter what it is, but the point is, it's going to write it to this logs folder, which is actually empty right now. It has a .gitkeep just for source control, but it definitely doesn't have a log file. In order to write it outside of the container, I'm going to use our -v syntax. Notice, I'm using PWD again. This is the PowerShell syntax, though, with the curly braces. And then I have var/www/logs, which is where it would try to write to from a code standpoint. So it will try to write to the container, but I want it to, in essence, redirect it to my current working directory here, which would be my host machine, in this case, my Windows machine. Now that would write it to the root of this project. That's not where I want it. I want it to go to the logs folder, so let me go ahead and fix that by saying /logs. So PWD will get me to the folder on my host machine, but then /logs will get me to a sub folder. I currently don't have a database running. We'll get to that, again, in the next module, so this is going to error out, but it should try to create the access log file. So let's go ahead and try it out. So I'm going to hit Enter. We'll let this fire up the container, and now keep your eye on logs. There we go, access.log just showed up. It's not going to have anything in it because I'm not able to hit it yet. But once we hook this up to a database shortly, then every request that's made to a browser will show up in an Apache type of log format, and it will show up in that file. That's a great example of more of a production scenario where your app needs to write to some directory, but we want that to be stored somewhere else. That's how we can do it. We can use volumes. Now I want to emphasize that I'm using a host volume here. Okay, the virtual machine or, in my case, the Windows machine that's running, it is actually the host for these containers, therefore, the logs are written to that host. What if the host goes down and is not recoverable for some reason? Well, you hope you have backups, but I want to emphasize that there are other types of Docker volumes that can get more sophisticated so that you can write to other network locations, for example, if you want it. Normally, this is a great way to start, though, and then, if you want to take it to the next level, go check out the Docker documentation on that. So that's an example of a development scenario using volumes and now more of a production-type scenario using volumes. So let's finalize this module and do a quick review.

Summary

This module's been all about containers and how you can get them to run. So the first thing we reviewed was using docker pull to pull from a registry down to your machine or to another server. From there, we looked at the docker run command, how to define things like ports, and the image that you want to use. And once a container's running, we saw how we can access container logs using the docker logs command, and then you can provide the name of the container or the container ID. From there, we talked about more persistent storage. What if you have log files, database files, and things of that nature that if a container goes down or is stopped, you don't want to lose those, you want to store them somewhere else, such as the host machine? Volumes allow us to do that, and we saw how we can use the -v syntax in the docker run command so that we can define either a volume that docker manages, or, in our case, we used the pwd syntax, so we defined where that directory was on the host system. Now that we've seen how to run a container, let's talk about how multiple containers can be run and communicate with each other, because at this point we have a Node app that still needs to talk to a database. So let's take a look at how we can make that happen.

Communicate between Multiple Containers

Module Overview

In this module, we're going to talk about how we can communicate between multiple containers. We've seen how to get a single container running, but you're probably going to have others. You might have APIs, possibly a database, really depends on your setup, but in that scenario, what do you do to talk between these different containers? So in this module, we're going to focus on a few things here. First off, discuss what are the options for container communication? Then we're going to discuss one called a bridge network, and we're going to see how to run multiple containers in the network so that they can talk, but containers outside of that network wouldn't be able to talk to the others. From there, we're also going to look at a bonus troubleshooting feature, which is how to shell into a container. This can be useful when you have a container up and running, you need to get inside of it and maybe run a command for whatever reason, and I'll show you the reason we're going to do it for our sample app. And then finally, we're going to introduce another tool in the Docker toolbelt called Docker Compose. This is an orchestration engine that you can use to not only run multiple containers, but you can also use it to build multiple images. A very powerful tool to know about, and we'll introduce that tool at the end of this module. So let's jump right in and talk about what are the options for container communication.

Create a Bridge Network

When you have multiple containers that need to talk to each other, they can talk over a network. Now in this case, we're not talking about a network between machines, although there are different types available. We're going to talk specifically about a bridge network, and this is a way on the same machine to allow specific containers within a network to talk to each other. But as I mentioned in the introduction, if they're outside of that network, they won't be able to talk to each other. So let's say that you have an application that maybe has some front-end code. It also has some back-end APIs. Those back-end APIs then talk over to a database. And for the sake of argument, we're going to say that everything you see here is running in a container, including the database. So now we need a way to run the app in its own container. We have three containers for the APIs. We have a database container, and, of course, you might have others. You could have a caching server, and the list goes on, and on, and on. So how do we communicate between all these containers? We know how to run them. That's not a problem, but how do we make it so they can actually talk to each other? The way we're going to do it coming up is to use something called a bridge network that I mentioned. Now a bridge network is a way to, as you see here, create a bridge so that data can then flow between the containers. They can actually talk to each other and talk back and forth. As long as they have access to that bridge, they can talk. Now this is the easiest type of network to get going in Docker. There are others if you check out the documentation. But when you're on a machine that has containers running, this one is actually very simple to get started with, as you're going to see, momentarily. So let's look at the command we can run to create a bridge network. So in addition to the docker build and the docker run commands, there's also a docker network command, and it has a create. So the first thing you'll do, though, is define docker network create, then you're going to need to say what type of driver are we going to use? Now there's many types of drivers out there. You can actually do a true network where you can go across multiple machines. But a bridge, in this case, is what we're going to do, so we can communicate between containers on the same machine. Then finally, you're going to say, well, what's my network name? What are we going to call this network? Now I doubt you're going to call it isolated_network. That's the name I gave it here. I put that, though, because I want to be really clear on what this type of network does. A bridge network, again, is for containers that you want to talk to each other. Let's say you have five containers that all need to talk. Well, they would need to specify isolated_network, in this case. What if you have a sixth, seventh, and eighth container that shouldn't be able to talk to the first five? They wouldn't be in isolated_network, and I'm going to show you how we add them to the network in just a second as well. In addition to network create, there's other commands you can run, too. These are very simple commands you can run, and there's a few others. But to get started, you have docker network create, which we just looked at, and then you'll have to give it the command line flags. There's also a docker network ls command. We can list the networks. And so if you've created some, and I don't know, maybe you forgot the name of a network or something like that, this would let you get that name. And then finally, if you want to remove a network, you could say docker network rm, very similar to how you remove a container. And then you can give it the network name, and there's a couple options you can do there. So now that we've seen that, let's go ahead and jump over to the editor, and the first thing we're going to do is get a network set up. So coming back into the project code, we have our node.dockerfile. We've seen that quite a bit up to this point. At the top, though, I have some comments here. And I just want to point out, I do list two options. The second one I don't recommend, but you may come across it, which is why I want to mention it. The first one is what we're going to do. We're going to do what you see right here, which we just looked at a little bit earlier in the slides. Then we'll go ahead a little later, and we're going to add containers into this network called isolated_network. Now there is an older way. It's called legacy linking, and you might see this --link command. This isn't recommended anymore, but you may come across it, which is the only reason this is in here, So we're not going to go into that because it's the legacy way, but I do want you to be aware that if you see --link, that's kind of the old way of doing it. So the first thing we're going to do, as I showed, there's a docker network ls command, and we can actually list our networks. Now, these were created by Docker. I didn't have to do these myself. But what we need is a custom one because we want to run our app container, and we have a database, which we haven't looked at yet, but we're going to. So what I'm going to do is just grab this top command here, and we're going to create a network, and it's going to be that simple. We're going to say the driver is a bridge, and we're going to call it isolated_network, again, give it a better name than that, but this will be very clear that it is isolated only to the containers that are part of the network. So let me go ahead and do that. Alright, there we go. Now, let's back up to our doctor network ls, and I now have a isolated network, and it's a bridge network. So now on the same machine, we can have different containers running. Now that's how easy it is to get started creating a bridge network, but we obviously need to get the containers into it, so we're going to look at that in the next few sections.

Run a Database Container in a Network

In our sample app, we have a database container that we need to get up and running, but if I don't put it in a network, then my app container, which is Node.js, wouldn't be able to call it in the first place, so it wouldn't really do much. So let's take a look at how we can do that using the docker run command. So we've seen docker run many times up to this point, -d will run it in detached type mode, but then we need to add some extra information to run the container and the specific image we're going to list in this isolated network that we looked at earlier. So what we can do is add a --net flag and then there's that network, isolated_network. Now, the rest of this would be normal, you can define ports, but in this case we're going to give it a name, which we've looked at earlier as well. Now that name is going to be important, and we'll look at that as we talk about connection strings, and we actually run the app that calls this particular container that has this database. And then finally, we're going to have the name of the image, which is going to be mongo in this case. Now you may be wondering why I don't list any ports there, and that's because in this case we're going to default to whatever is exposed by the container itself. Mongo has a very specific port it runs on, and we're just going to leave that as the default. So let's switch back over the editor and we'll get this one up and running. So first off, let me clear my console. We know we have our network, it's available. So coming back into our steps, the next thing I'm going to do is run the command that you just saw. So we'll go ahead and start this mongo image and run it as a container. Now normally with databases, there's going to be quite a bit more you'll likely do. This is something that if you work with a DBA, you'd have to talk with them. Some companies choose to run their databases and containers, others still leave them outside of containers. So it would completely depend on your organization, but it is absolutely possible, I do it, I've done it for many years now, and it's actually worked out really well because I can update the database simply by getting a new container up and running. Now what we're going to do with Mongo then is run it. If the image doesn't exist on your machine already, then of course it's going to pull down the layers to your machine, I already have it. It looks like it gave us back a container ID, so let's see the status of it. We'll do docker ps, which is only running containers. And there we go. So it gives the command docker-entrypoint.sh. We have the status, it's been up for a little while now. And there's the port I was talking about, 27017, that's kind of the default for Mongo. And then notice the name, that's going to be important in the next section as we start talking about the database. Now that's how easy it is to get this up and running in that network. We put isolated_network with a --net flag and we're ready to go. So now that we have this database up and running, let's go ahead and switch to the container for the application.

Run an Application Container in a Network

Running the application container in the network is going to be identical to running the database in the network. So let's take a look at that command again. We're going to do docker run -d. There's the ports we're going to use in this case, 3000 on external and internal. And then, just like you saw earlier with the database, we're going to put it in the same network, isolated_network, in this case. Now from there, we're going to have the name again. Name, in this case, would be optional, but it's going to be nice to have. And then finally, we would have whatever our image name is. You can see it's very simple once you know some of these different flags. So let's switch back to the editor again, and we'll take a look at getting this container up and running and hopefully talking to the database. So let me clear my console again, and we're going to do the same type of process. Now you'll notice here, it's a little bit longer, but I'm going to go ahead and paste this in. So we have the detached. There's the key part, at least for this module, our isolated_network. We give it the name, there's our ports, and then we have the volume that we looked at earlier for logging and things like that. Now I'm going to run danwahlin/nodeapp, and I'm going to run the 2.0 tag of that because I don't have a latest on the machine. All my images are tagged with versions. So I'm going to go ahead and hit Enter, and there's our ID. Now let's check it out. We'll do docker ps, and looking good so far. We now have two containers running. So we have our nodeapp, and you can take a look at the names over here, and then we have our mongodb. Now, before we go to the browser and actually try this, I want to circle back to the name that I mentioned was so important, at least for the mongodb name. If I open up the app's config folder and I go into config.development, which is what we're running right now, look at the name of the host that's configured, and then, of course, we have the database. But the host, that name, matches up with this name. Very, very important. Because the way we're running containers right now, we don't control the networking, IP addresses, things like that, but we can control the name. Now localhost wouldn't work in this case because that would assume my machine, not the container. So by putting mongodb as the host, the app itself, when it uses the connection string, and you can find that in the lib folder, go into database, and there we go. We get the connection string right there. Notice it's mongodb. There's the protocol. Config.host is what I just showed you, mongodb, and then config.database, that's our funWithDocker. So it's actually using this name that we defined for the Mongo container, and that's why I emphasized it's so important. Now did it work? Well, let's open up a browser and try it out. So I've navigated to localhost:3000. This is looking pretty good. It's kind of hard to tell, though, because guess what? We don't have any data in the database. We're going to fix that in the next section, but at this point, it's looking good. How do we know for sure? Well, let's go back to the logs, and we can see if this is actually up and running and is it actually connecting to the database? In an earlier module, I showed you how you can get to the logs of a container, and this is a really important troubleshooting-type task or, just in this case, informational-type task. The first thing we can do is let's clear and let's do docker ps again and see did the container crash? Because it could've. Nope, looks like it's been up for 3 minutes. That's good news. Now I'm going to take the ID. Looks like aef would be enough, so we'll say docker logs aef. Alright, and let's go ahead and move this up, and look at that. That's good news, db connection is open. Don't have any errors like we had before, so it successfully connected to the MongoDB database. Very nice. Now we have two containers running in our network. If we go back to docker network ls, there it is. And then just to show you some other commands, if we do just docker network and hit Enter, notice we can also inspect. So let's go to our isolated network, b87, and we'll do docker network inspect, and then we're going to do b87. Now this is going to be a big dump of all the data, but look right here at the bottom. There's our containers in the network. There's our nodeapp, there's our mongodb. So it looks like we're in good shape. They can talk to each other, but containers outside of this network would not be able to talk to each other. Now everything is looking good at this point, but we didn't have any data. So, in the next section, I'm going to show you how you can shell into a container if you ever want to run a custom command.

Shell into a Container

Another test that you may need to run from time to time is shelling into a container. We're used to running different types of commands on our system, but how do you run commands in a container? So to shell into a container, you can use the docker exec command as you see here. You have to add a command line switch, -it, and this stands for an interactive tty, basically an interactive terminal. And then you're going to give it either the container name or the container ID. And then finally, you can give it the sh shell. So if we break that down again, we have an interactive tty it's called. It's kind of an old term that goes way back. We're going to go ahead and give it the shell to use here at the end, which is sh in this case, but the type of shell you use depends on the container. In the Linux world, it may be sh, could be Bash, could be something else. In Windows, it may be PowerShell. So you'll have to look at what your container supports there when you do this. And then, of course, the container we go into is the container ID or the container name that you see there. So let's look at an example of how we can use this command. I've started up the node container, and I've started up the database container. I'm going to come down to the browser again. Let's go ahead and go back to our localhost 3000, and there we go. Now we didn't get any commands though. So the problem is there's no data in the database. So let's go ahead and fix that by shelling into the container and running a dbSeeder that I have in the project. So coming back in, you'll notice that in this case we have our danwahlin/nodeapp:2.0. We have an ID of 114 there. We have a name of nodeapp. Now you can also get to the information about the running containers by using the Docker extension that we looked at in a previous module. I can come on down and I can even drill into this right here, and now I'm into the file system of that container. Now that's kind of the easy way to shell into a container, or get inside of it, is to go and use the Docker extension. Now I'm going to show you how to do it for real though on our own. So let's just come on back to the file system. And I want to point out that there's a little dbSeeder right here, and all it does is seeds the database with some data. So it jumps into the lib folder and it runs a dataSeeder. And the dataSeeder just adds some of these

Docker commands that we're learning, like docker run and the imageName shows an example, and some others, and it adds that into the MongoDB database. Now we need to run that, but I can't run that from here. I can't run node dbSeeder.js from here because that's on my system, and this is actually running in containers. In fact, we have two containers, as you'll see, one for Mongo and one for Node. So what we're going to have to do is shell into the container. So we can do docker exec. We're going to do - interactive tty, interactive terminal you can think of it as, and then we could either give it the container ID, 114 in this case, or I could say nodeapp. And now I can give it what shell do I want. Now there's a lot of shells out there, and it really depends on what your container uses, but a lot of times you'll go with an sh shell. And this is a Linux concept, in this case, that you can use. So I'm going to hit Enter, and now notice I am in the working directory. If we look back at our dockerfile, our working directory was /var/www. All right, well that's where it put me. So I'm going to do an ls command, and there's my files. Now these are the files again that were in the image and now in the running container. And you'll notice that one of the files here is the dbSeeder, right down here in the left corner. So what I'm going to do now is run node dbSeeder. Now I'm in Linux at this point, so it's very, very case sensitive, as a heads up. So we'll go ahead and do that, and there we go. It connected to the database, and it looks like it seeded it. So I'm going to do Ctrl+C to stop this. Now I'm back here, still in the container. Now we could actually hop around this container. If I want to move up a folder, I could do cd.., do an ls here. There's what folders we have. I could cd back into the www. There we go. And I could even cd down into my other folders, like public or whatever you'd like to do. So it's a great way to be able to navigate around, and you can even edit files here. I could use vi, or nano, or some different editors, whatever your container has available. So now that we've seen that, let's get out of the shell. We'll type exit. Now we're back to our local system, and we're good to go. Just let me clear this. Now let's go back to the browser and we'll hopefully see that seeded data. So let's hit Refresh, and there we go. We have our run command, we have our ps command in this case, and we could even come in and add commands if we'd like. So that's an example of using docker exec. Very useful any time you need to get into the container and then interact with the folders, the files. Maybe you just want to open a log that's not part of the Docker log, it's a different log, for example. Maybe you want to edit a code file. Now normally you're not doing that in production, as a heads up. That would typically be a development-type exercise, but it's a good command to know about.

Building and Running Multiple Containers with Docker Compose

So we've now seen how to run multiple containers in an isolated network, a bridge network, but you saw that you have to run several different commands to make that happen and get it all working. What happens if you do have 5 or 10 or more containers running? That's a lot of commands. Sure, you can batch them all up, but what if there's another way? So in this section, we're going to take a real quick look at another tool available with Docker called Docker Compose. If I had to pick my favorite tool in Docker, it would probably be Docker Compose. Now, obviously, you need just the core Docker functionality for building images and running containers, but once you get all that, as a developer, or even in other scenarios as an admin, you may need to bring multiple containers up simultaneously. You might want to build multiple images and do those types of tasks. So Docker Compose can do those things. It's like juggling, like the octopus here, multiple containers or multiple builds. So let's walk through a few of the key features. First off, Docker

Compose defines something called services using YAML configuration files, and if you're not familiar with what YAML looks like, we'll take a quick look at that. A service really equates to a container at runtime. You can also use Docker Compose to build one or more images, your services will define that. You can start and stop one or more services using Docker Compose commands. You can view the status of running services, and you can even stream the log output of running services. Recall earlier the docker logs command. Well, this can do the same thing, but with multiple containers streaming the logs out to you in the console. The way it works from a workflow standpoint is you can build services with a command, you can start up services with a single command, and you can tear down and basically remove the services with a single command. And once again, services are really containers in this case, but they call them services in the Docker Compose world. So the way it works is you will create a docker-compose.yml file, like you see here. In this YAML file, the first thing that you'll normally see at the top is the version, and there are different versions, they change over time, of course. But the way this works is you put a property name on the left, you separate that with a colon, and then you put the value on the right. It kind of looks like JSON, JavaScript Object Notation, if you've seen that, but you're going to see there are no curly braces or anything. In fact, this is really a higher level form of JSON, technically. Then you're going to have your services container. In this case, services, and we're going to come back to that, but you can also define networks. Earlier we talked about how a bridge network could be used to run multiple containers in that network and then have them communicate. We're defining that we're going to have a single network, it's called nodeapp-network, and it's going to have a driver of type bridge. So we're basically creating a bridge network. Now what containers are going to go in this network? Well, those go under the services property. I have one called app, and then I have one called db, and it's also in the same network. Now where you see the ellipses there, that would be additional properties, you could specify things like where is the image that would be used to run this, and there's a bunch of other settings you can define. So now we have the same setup as before, but instead of doing it imperatively on the command line, we're doing it declaratively in YAML. And then I'm going to show you some commands you can run to basically bring this YAML to life and either build your images or run your containers. Now, one thing I want to point out here before we move on is that with YAML, the indentation absolutely matters. You'll notice that I have app in two spaces here. I have db that matches that level. Notice networks is the same level as services and version. That all matters with YAML, and when you're really new to this, that's a little bit painful, I'll admit, at first. To be honest, it's like anything in programming, though, it just grows on you over time and you kind of learn what to do and what not to do. It takes a little bit of practice. Now obviously, this file on its own is pretty useless, so we need some commands in order to do things. The key commands we're going to be looking at coming up in the demo here are, first off, docker-compose build. That will take your Docker Compose YAML file, and based on the services defined there, in our case we had two, let's say that only the app, though, had a custom build. Let's say the database is from Docker Hub, that image, such as MongoDB, and we don't have to build it, it's just used. Well, then this would build our single image. You can build many images, though, using this command. Now we don't have to do a docker build command like we learned earlier in the course. We can just do one command to kind of rule them all. Now the next one is, once you have the images available, you can, of course, run your containers, or, in our case, run your services. We had two services earlier. We had the app and the database service. To run those two containers and bring them up, we can do docker-compose up. And then finally, when we're done, we can bring everything down and delete the containers with one single command, docker-compose down. So you might wonder, okay, what should I do then? Do I do docker build and docker run commands, or do I use these types of commands? Well, for me personally, based on my previous experience, once I learned those commands, you still need to learn them and know them, but I almost always would use docker-compose. Why would you want to run docker build five times if you have five different images to build when you can make a YAML file once and then run docker-compose build? It's much easier. So now that we've talked about some of the basics of Docker Compose, let's go ahead and dive into a demo and I'll show you some of these different commands in action.

Using Docker Compose Commands

At the very beginning of this course, I showed a quick demonstration of how to get the sample app up and running, and showed some Docker Compose commands. Now in the last section, we talked about some of those commands. What I'm going to do now is walk you through a little more detail on the YAML, how that all works, and then we'll go ahead and use those commands that we saw at the very beginning to wrap this course up. Coming back into the sample app, I have a docker-compose.yml file. You're going to recognize what we just covered in the previous section here. We have a version, there's our services, and then we have networks with a bridge network called nodeapp-network. The content of the services property is where all the important things happen. So we have two services, we have a node service and we have a mongodb service. Now notice we have a container name, and container_name is actually a property of the YAML for Docker Compose. This is something you can look up in the docs. We have an image, and that's going to be the image name, nodeapp. And then we have a build context. Now, what this is doing is instead of having to do a docker build command, you can just run the docker-compose build. That will look in this file, look in the current folder, so we're in the root of the folder here with the dot, and then it will find the node.dockerfile that we've been working with pretty much the entire course. So when you run docker-compose build, which I'm going to do in a moment, it will actually use that and build this particular image. Now, of course, once that image is built, you could retag it, rename it, those type of things, but it will make it really easy to build an image, and, of course, you could have many of these in a Docker Compose file. So with one command, you could build pretty much as many as your system can handle images. Now moving on down, notice we have the ports. We have, just like we've done before with docker run, we have the external port of 3000 and the internal port of 3000. Here's the network. That'll make sure that we're part of that bridge network. And then it depends on mongodb. Now that name matches with this property here. What depends_on will do is it'll ensure that MongoDB starts first, but it won't make sure it's ready. So an important thing you'll have to do with your apps is if you have, for example, a web app that's going to call into a database, you'll have to make sure the database is available. If you get an error, you'll have to retry if that web app tries to hit it right at the beginning. So that's important to note, that depends_on will ensure that the MongoDB database starts first in the container, but it won't make sure that it's ready to accept connections. That'll be on whoever develops the app container to add code to handle that. Now, notice that the mongodb container itself just has a container_name, mongodb, and an image, but it doesn't have the build with the context and the dockerfile. So what's going to happen is it will look locally for an image called mongo. If it can't find it, it'll go to whatever's registered as the default on your machine for a registry. In our case, it's going to be Docker Hub. And then finally, we see it's in the same network as the application service, node, up here. Now, just like I said earlier, where the name mattered, the name right here really matters. The service name ultimately will become a network addressable name so that the connection string will work. And recall earlier that I showed you config.development, and that had the host of mongodb, and we can also open up the production, which looks the same in this case. That's going to be the name that you see right here. So if you want to change that name, you'd have to change the connection string. There's a lot more you can do with Docker Compose. You can certainly bring up many other services if you want, and I'd recommend checking out the next course in this learning path, which is going to be on Docker Compose, and provide more detail if you'd like to dive in deeper. With that, I'm going to go ahead and just right-click, open a terminal. And the first thing I want to show you is if you type docker-compose, and you could just hit Enter, but I'll type --help, there's several different commands you can run, you'll notice here. And we're going to focus just on the main three, but there's also a bunch of command line switches, and what I want to point out is -f. If the name of your docker-compose file is a different name than docker-compose.yml, then you could actually put docker-compose -f, give it that file name, and then say build. That way you can actually change up the file names if you'd like, because who knows, you might have a development version, a production version, things like that. Now for this demo we named it the same as it expects. So we'll say docker-compose build. Now this'll kick off the build process for our node service, but not our mongodb, because mongodb doesn't require a custom build. So let's go ahead and do that. And there we go, most of the layers were already cached, so we're kind of good to go there. So that's the first step. Now if I want to run this and get the network set up, the two containers running, and all of that, then we can do docker-compose up. This will now look at the services, there we go, our mongo came up first. Now our node is trying to connect, and it's now listening on this port, and it actually looks like it all came up very fast. So let's go ahead and run to the browser. We'll go to localhost 3000, because that was the external port, and there we go. Our app is running, again, we don't have any data in the database because we just started it up, but we could either integrate the dbSeeder script into the startup of the app or we could shell in as I showed earlier and we could run the dbSeeder script. Now going on back, let me open another console, and I want to show you that under the covers it's just regular Docker. They're just hiding some of the complexity. If we do docker ps, there's my two containers. So what do we do from here? Well, the last command is down, so let's go ahead and do docker-compose down. Alright, everything is now gone. Kind of prove that by doing docker ps. Okay, nothing there that's running, but is there anything behind the scenes stopped. Docker ps -a, nothing. So you can see how easy it is to not only build from our dockerfiles, but also run our containers, create the network, and then take everything down, and really all you have to know is three commands. Now from here there's a docker-compose logs. If you want to stream the logs, you could do that. Take a look at the help there that I showed earlier if you want more information. Now the final thing I want to show you is we can actually right-click on this. If I scroll down to the bottom, notice there's a Compose Down, Restart, Up, and Up - Select Services. Now I'm going to go ahead and do Compose Up, and let's watch what happens here. And there we go, look at that, it rebuilt all the images that need it, and then it started up our containers here. And then it says this terminal is going to be reused by tasks, press any key to close it, so we'll close it. Let's do docker ps -a. There we go, everything's been up, and we should still be able to hit it in the browser. Now from here, let's right click on it, come on down, and we can say Compose Down. That will issue the command. There's that -f switch, by the way, right there. And we're all done, we'll hit any key, let's do docker ps -a, everything's gone. So if you want, you can also use the Docker extension for VS Code that

I discussed earlier in the course, and you can do some pretty cool things there as well when it comes to the build and the up, and then taking those containers down. So that's a look at Docker Compose. Now I want to emphasize that there's a lot more you can go into with Docker Compose, so, like I said earlier, check out the next course that'll be in this learning path if you'd like more details. So with that, let's go ahead and do a wrap-up for this module.

Summary

In this module, we talked about bridge networks, how we can set those up for containers so they can communicate, and we also talked about Docker Compose. So as a quick review, we started off talking about what a bridge network is and how on a given Docker machine, it will allow us to communicate between containers that are in the same network. We use the docker network create command to do that, we specify the type of the driver, which is bridge, and then we can give that network a name. Once we have the network, we can use the doctor run command along with a --net switch to define the name of the network that that container should run in. We saw how you can bring up a Node app container that way, as well as a MongoDB container that way. Finally, we took a look at Docker Compose and saw how it's an orchestration engine and that we can use it to not only build our different images we need, but also to run containers. This is all done in the YAML file, docker-compose.yml, and that's where we can define our different services, and if our service, which ultimately becomes a container at runtime, needs to be built as an image, we can define all that type of data in the Docker Compose. Now it's super easy. We can do docker compose build, docker compose up, docker compose down, and other types of Docker Compose commands. So that wraps up our discussion of building and running your first Docker app. We've now looked at all the core concepts you need to know to get started. Are there more? There's always more. The learning never ends, but you now know the core things you need to know. You know how to build images, you know how to push those to a registry, pull them, run them as containers, communicate between containers, and even orchestrate containers using Docker Compose. So I'd encourage you to go through the rest of this learning path. You can also check out my Docker for Web Developers if you'd like this type of information but in a more detailed format. So with that, thank you so much for joining me. I hope you've enjoyed the content, I know I enjoyed putting it together for you, and I wish you the best in your journey working with Docker. It's a lot of fun to work with and a great technology to know.