Course Overview
Course Overview

All right then, Docker. It is changing the world and in a good way. And believe me, you don't want to miss out. Well, let me tell you, have we got a course for you. Getting Started with Docker is the ultimate launchpad for kickstarting your Docker journey. And it's handcrafted to be perfect for developers, operations professionals, and of course DevOps. But the best part, not only does it take the scary out of learning Docker, it actually makes it fun. And when you're done with the course you will be pumped and ready to learn more. So, in this course, and there'll be plenty of opportunities for hands on, but we'll show you some really easy ways to get Docker. Then, though, the beating heart of the course is showing you how to take application source code from the GitHub repo, build it as a Docker image, share it in a container registry, and then run it as a container. And you'll get your hands on with tools like Docker Compose and Swarm and Stacks that all make it easy to manage microservices apps in a modern, declarative fashion. And of course we'll explain what all of those buzz words actually mean. Now, what about me? Well, I'm Nigel, and I've been working with Docker in containers since the early days. I'm one of the longest-serving Docker captains, I'm the author of these bestselling container books, and of course I've got a ton of courses here on Pluralsight and the reviews speak for themselves. But you know what, let's cut the waffle, and let's get learning Docker.

Course Introduction

Course Introduction

Alright then, here we are. So thanks for choosing the course, and I'm telling you, it's a good choice. You're going to love it. Now, some really quick housekeeping before we start, and I promise it will be quick because I'm hoping you won't skip it. So the whole reason for this course, well, obviously it's to get you up and running with Docker, but we're going to do it as fast as possible, but without leaving your brain fried and totally confused. So, there'll be plenty of chances to get your hands on, but we'll be doing everything we can to explain things as clearly as possible, because believe me, right, I have no interest in you walking away at the end of this course, and I don't know, maybe recognizing a few of the commands, but inside your head, you didn't really grasp what was going on. No chance, right, I have no interest in that. We will be explaining things. However, we've got to keep in mind, this is a beginner's course, so even though, yes, we'll be explaining stuff as we go, we're not going to be getting into the weeds. In fact, we've got other courses for that. And you know what, actually, on the topic of other courses, well, if maybe you are like 100% a bona fide nube, like, I don't know, you have zero idea what Docker is, and a container is something to hold a beverage, well, if that's you, you probably want to hit this course first. It is a proper, absolute zero prior experience necessary. Anyway, look, the stuff in this course here is the fundamentals, and Docker itself, at its core, is a DevOps tool. I mean, it was instrumental in the whole early DevOps movement. Well, because Docker is a DevOps tool and we are learning the fundamentals, everything on the agenda has both a developer and an operations element, or it applies equally to both, which is why we've stuck this course right at the start of the developer and the operations learning paths. Now, we're building these paths out, and I don't know, I expect that we'll constantly be tweaking them, but the point is, we've got a

developer learning path and on operations learning path, and this course is the springboard for both. So, one of the things I'm aiming for is to wrap your head around Docker and containers and what they're all about, to give you a chance for some hands-on with some of the cooler aspects, but ultimately, okay, to lay the groundwork for you to crack on with the other courses in the paths that are right for you. However, and this is good, right, as well as the learning paths we've also got standalone courses that I know you'll love as well. Now, just a couple right, on the developer side, Dan's got a great course called Docker for Web Developers. Then on the Operations and DevOps side, I've got Docker Deep Dive. And, look, I could waffle all day, okay, but you know what, we're adding courses all the time, so check out the learning paths, for sure, but also poke around in our standalone courses as well. You'll just love them. I don't know, maybe take a note or maybe a screenshot of this or something, cause I really want you to maximize what you get out of Pluralsight as a platform. So, whatever you do, do not stop after this course. It is literally the tip of the iceberg. Anyway, look, who am I? Well, like I mentioned in the trailer, I'm Nigel, and I know some of you know me already, but for those who don't, I'm a bit of a techaholic. And I've been working with Docker and Kubernetes for, well, let's just call it a lot of years, yeah, like when Docker was really, really new. In fact, look, I'm a Docker Captain, though, anybody referring to me as captain will be ordered to walk the plank. Look, I've got a ton of courses here at Pluralsight, I am the author of these bestselling books on containers, and I'm pretty active in the community and you'll see me at conferences and meetups and the likes. Well, actually, speaking of which, if you do see me anywhere, and I say this all the time, but I do mean it, if you see me somewhere, come and say hi. Like, I might look miserable and unapproachable, but I'm not. Come and say hi. Oh, of course, let's connect on the socials as well. I'm always happy to engage and talk about technology. But, look, talk is cheap. This is a cracking course, and I know you're going to love it, and I know you're going to love Docker as well. So, strap yourself in, and let's get ready to learn.

Getting Docker

Module Overview

Okay. I want everything we learn to be as hands on as humanly possible. Now, in saying that, if you can't follow along in your own lab or, I don't know, maybe you just don't want to, that's all right as well. You will still get a ton of good stuff from the course. I guess all I'm trying to say is that I reckon for maximum take-home value it is best if you can follow along. Anyway, look, if you do want to follow along you are going to need Docker. And let me tell you, getting Docker has never been easier. So, gone are the days of me literally robbing 30, maybe even 40, minutes of your time just showing you how to install it. I reckon today like, I don't know, 5, maybe 10, minutes, right, and we will be rocking and rolling with a Docker environment that you can totally follow along with. So, we'll look at two options, Docker Desktop and Play with Docker. Now, look, the aim of the game here is to get you something quick and easy so that you can follow along. We're not aiming for battle-ready, highly-available, high-performance, weapons-grade Docker clusters here. No, we've actually got courses for that. But here and now we are getting you a lab environment and a pretty good one as well. But you know what, time is money. Let's get this show on the road.

Docker Desktop

Right then, Docker Desktop, and for me, this is the easiest way to get yourself a Docker development environment on your laptop. And I'm telling you, it is slick as heck. Well, it works on PC and Mac, and with a few simple clicks, you're going to be in business. However, as things stand, if you're a PC user, you're going to need a 64-bit version of Windows 10 or later. If you're on a Mac, I think the official line is something like the most recent version of macOS, plus maybe the two previous. But you know what? Never take my word for it. Always check the docs. Now as well, while this is about getting you a lab as a local testing development environment, you're actually getting the full Docker experience. Pretty cool, right? So it is brilliant for, let's say, developers coding your apps, as you do with whatever tools you have on your laptop in your favorite languages, only now you're adding the tools to build those apps as Docker images and then test and run them as containers, and because of the way that containers work, if your app works on Docker Desktop, it is pretty much nailed on to work in production. But you know what? I could talk all day, and we don't want that. So let's go and get it. Now I'm here a docker.com, but of course you can just as easily Google Docker Desktop, but I go products and Docker desktop. Ah! Now it's detected that I'm on a Mac, but look, it's also got Windows. Oh, and look, even a link to Linux. Now, right now, they do an Edge under Stable channel. Stable is the safe route and probably more like what you'll be running in production, whereas Edge, that's more for the risk takers, and it gets you some of the newer and maybe more experimental features. So for me, I'll go for Edge. Now, while that downloads, okay, all of this is pretty much identical for Windows. It's basically fire up the installer and then next, next, next until you're done. Well, I'll tell you what. Here is mine in my Downloads folder. I just fire that up, and okay, it's going to do a bit of thinking, and then yeah, look. It's just drag over here to my Apps folder, and away that goes. Now of course, depending on your internet connection, this might take a minute or something like that, but when it's done, it's then just like any other app in your launchpad on a Mac, or I guess, in your Start menu on Windows. Oh, yeah, customary security stuff. And then on a Mac, you get a whale at the top here. For Windows, it's a whale in the bottom right-hand corner in your system tray, but once it's all settled and it looks like this, that's Docker running. And seriously, at this point, you are ready to rock and roll with a full-on Docker environment on your local machine. But I want to show you some quick stuff, right? So clicking on the whale gets you a bunch of options, and now look, you can restart it. There's Kubernetes stuff. Now look, yours might be empty, by the way, your Kubernetes stuff, but we'll have a look at preferences. Now, okay, the General tab lets you say things like, I want to automatically start when I boot up. Obviously, I probably want to check for new versions or stuff like that, right? Resources is interesting. It lets you keep a leash on how much of your system Docker consumes. I'm going to skip Engine and CLI actually, because we're a Getting Started course, but Kubernetes here, this lets you run a local Kubernetes cluster as well. Now, okay, we're getting started with Docker, and I don't want to get off track or confuse things, but if you are needing to test and develop against Kubernetes, boom! Right there, Docker Desktop has you covered, as does Pluralsight actually. I don't know, maybe take a screen grab of this or whatever, because this is the best way to start your Kubernetes journey. Well, a couple of last things before we crack on. Docker Desktop on Windows can run Linux containers and Windows containers, which, when you think about it, is pretty sweet. I mean, you can use your Windows laptop to develop and test Linux and Windows apps. In fact, you get a nifty little option like this that lets you flip between the two. Unfortunately, Docker Desktop on Mac, yeah, that only does Linux apps. Anyway, magic! If you've got Docker Desktop, you're ready to, well, do you know what? Not

only follow along with the examples, but you're ready to do so much more, because like I said, it is full-on Docker, and you know what? It's becoming more and more popular as a local dev and test tool.

Play with Docker

All right, Play with Docker. So if you've been following along you'll have seen how to install Docker Desktop on your own computer. And it's the business right, I love it, but what if you can't install software on your computer? Or, like, maybe you've only got access to a work machine and it doesn't let you install software. Or, I don't know, maybe your computer's to weird or something. No sweat, it's all good. Play with Docker has got you covered. All you need is a modern browser and a Docker Hub account. So, let's get straight to it. The URL's here, labs.play-with-docker.com. Okay, and as we can see, there's a cheeky little log in, but that's all right. All you need is a Docker Hub account, and they are free. And do you know what, honestly, if you're serious about learning Docker a Hub account is pretty much mandatory. So, if you haven't already got one, crack open a new tab and head over to hub.docker.com. Then just go through the sign up here. You know what, it is literally those three or whatever fields and you'll be in business. Anyway, once you've got that it's back here to Play with Docker and just whack in your credentials. Now then, and I love this, right, so don't get me wrong, but Play with Docker is provided to you courtesy of Docker, Inc. for free. So, if sometimes it's not the fastest or most responsive in the world, I don't know, cut them some slack. It's not like you're paying for it. Anyway, look at the clock up here. It is a 4-hour countdown. And I'm telling you, and trust me on this, when that clock hits 0 it is goodbye to anything you built. So, you are getting here a time limited 4-hour playground. But it is as easy as clicking ADD NEW INSTANCE here. Honestly. And that's it. That right there is a fully-working Docker instance. So, if you can't install Docker Desktop and you don't have access to a lab or anything come on over to Play with Docker and you can still follow along. And you know what, for us on this Getting Started course I reckon we are ready to peel back the curtain and see the magic.

Deploying a Containerized App

Module Overview

Okay, so no messing about. We are about to take an application all the way from source code to running in a container, plus all the funky stuff in between. Now the idea is you're going to get a proper hands-on experience that, I don't know, lights up the old neurons and creates one of those ah-ha moments. You know the type where you're like, okay, yeah, now I get it. Now I know what all the fuss is about and why I need Docker so much. But you know what, as well as that, I'm kind of hoping that the whole process we'll go through will give you this sort of end-to-end holistic view of what the major pieces and moving parts are, and yeah, both from a developer and an operations perspective. Now, a couple of things. Like I said, you're totally welcome to follow along on your laptop or wherever and get your hands on, but you don't have to. I am more than happy if you're just to watch, you're still going to learn a ton. The other thing I wanted to mention is that all of what we're about to do is 100%

relevant for developers and operations professionals. So, on the developer front you'll see how to package your app as a container image, share it in a registry, and then how to run it as a container. And you know what, you'll be able to do all of this on your laptop with crazy cool tools. Over on the operations front, well, you'll get familiar with Docker in general, like I mean what the heck are things like images and containers, but you'll also see how to run a container, start it, stop it, all of that jazz, right? And I reckon if you're familiar with virtual machines you'll see a bunch of parallels, yeah? But you'll also see a bunch of differences. Oh, and of course, if you're DevOps or if you're looking to get into DevOps, haha, throw me a high five because you are going to get a piece of everything. Well, either way, okay. In my opinion, vast experience, yeah, the best developers and the best sys admins or operations people all have some understanding of the full picture. So, if you're, let's say a sys admin, it is absolutely a good thing for you to know the fundamentals of what developers are up to. And then on the flip side, if you're a developer it's going to make your life a whole ton easier knowing some of the things that operations care about. But you know what, look, I'm waffling. Just trust me when I say the simplicity and slickness, if that's even a word, of what we're about to do, I am telling you it's going to blow your mind. Well, look, waffle, waffle, waffle. Here's the plan, we're going to race through a full workflow and we'll do at, I don't know, something like warp 6 or 7. A bit of a reconnaissance run actually. The idea being to give you a big picture so that when we drill into it a little bit later you've got an idea of how everything fits into that bigger picture. Well, do you know what, once we're done with that we'll then step through things a bit slower. I don't know, impulse speed, yeah? Now the idea with this bit is to reinforce everything that you'll just have seen, plus it'll give you more of a chance to digest things. And that's the plan. And like I said before, look, follow along if you want. Everything we're about to do can be done in Docker Desktop or Play with Docker. Whew, sound like fun? Well I'm telling you it is going to be, right? Let's get this show on the road.

Warp Speed Run-through

Okay, time for that reconnaissance flight. And remember, the aim here is just to grasp the end-to-end flow, so it is absolutely fine, in fact, I expect that some of you are going to get lost at some point, but it's totally okay, because this is like a movie trailer, or whatever, where you're just getting the highlights. Remember, we're going to be playing the full-length movie right after. Now then, this here is the workflow we're about to do. We'll take some application source code, and then we'll use Docker to package it as a container image, push that image to a registry, and run it as a container. And you know what, actually, for this part, even though it's demos, you should just watch. You can follow along in your own lab when we step through things a little bit slower later on. Anyway, look, as we can see, I have got Docker running. Now, this is actually Docker Desktop on my Mac, but it doesn't have to be. Docker is Docker, so it could be Docker Desktop on Windows. It could be Play with Docker in the cloud, pretty much any Docker installation, to be honest, But then this here is some source code, and I don't want you to worry about what that is right now. It is just source code for an application, though, actually, I will say, we've got a special file here called Dockerfile that tells Docker some important stuff about the app. Anyway, look, detail to come. With all of that in place, we can build that code into a container image. So, the command to do that is docker image build, and we'll talk about the options later. But away that goes, basically pulling the app code and building any dependencies and packaging it all as a neat and tidy image. Then

when that's done, we end up with this image here and no problems, actually, if you're unsure what an image is yet, we'll circle back later. But once we've got the image, the next step is to push it to a registry, which we do with this command here. Give that a second to upload. Okay, that is now on dockerhub here. Yeah, look, there we go. So, it's on dockerhub ready to be pulled and used. So remember, that's the app and all dependencies packaged up as an image and stored in a centralized repository. And then to run it, we just go docker container run here with a few parameters, and that should be running, which it is. And because I know that it's a web app on Port 8080, and I guess because I know a thing or two about Docker Desktop, I know I can reach that on local host 8080. And there it is, up and running as a containerized app. How about that, eh? We started out with some code and a list of dependencies, and then we've used Docker to package everything up as this cool image. We used Docker again to push that to a centralized registry, and we used Docker again to run it as a container. And I don't know about you, but I still remember for me, like the first time that I did that, I was blown away, like thumping my chest and feeling properly large and in charge. But, I get the flip side as well, right? It can kind of be a bit underwhelming. I mean, I don't know, it was almost too easy. Well, if you think that's the case, get used to it because that is the beauty of this. Simple is the new normal with Docker. Anyway, look, that's the reconnaissance flight. Time to look a bit closer and give you a chance for some hands-on.

Containerizing an App

Okay, so we're going to step through exactly the same workflow that we've just seen, only this time, a little slower. Give you a chance, maybe, to follow along and figure things out a bit better. Now, I'm going to be doing the demos in Docker Desktop on my Mac. But do you know what? Docker is Docker, and that's part of the beauty. So if you're following along, maybe in Play with Docker, or even Docker Desktop on Windows, or Docker somewhere else, it's all going to be good. Well, first things first. The app we're building is a Linux app. So, if you're on Docker Desktop on Windows, be sure you've got it rigged for Linux containers. If you're on Mac or Linux, you're set and good to do. Anyway, this here is the app, and it is publicly available on GitHub at this URL. Now, again, Docker is language agnostic. So we're not bothered about the detail here. I mean, yeah, I'll quickly walk through it, but it literally could be any app; Docker doesn't care. So, pick your favorite language and packages; they're all good with Docker. Anyway, we'll look at the main app here. Yeah, super short, right? It is an Express app using the handlebars view engine, and, well, you know what? Not massively bothered about the rest, other than maybe this bit here. The app itself listens on port 8080. So, that means when we run it in a container, it is going to bind to port 8080 inside the container. So, park that for later, right? Port 8080. 0h, and it'll print this message, but that's basically the app. Then back here, this file will list the dependency, so the packages the app needs, and under here is where the view lives. Anyway, look, a super simple web app that, if you're a front-end web developer, you can no doubt see, well, look, it's about as bog-standard as they come, isn't it, so the point being not written in any special way or special language to make it work with containers, and that's the whole point. Take pretty much any regular app, ha, it'll work in a container! Now, the process is what we saw before. Take the source code and build it into an image. Now, Docker is clever enough to do that, or it's almost clever enough. It actually needs a tiny bit of help, which is where this file here called Dockerfile comes into play. And I mean, look at it. It's basically a set of build instructions. So, a set of steps for Docker to follow to build the app and its dependencies into a container image. So all

the container image is is app code and dependencies, all neatly packaged so that we can share it and run it. Anyway, look, this is a Linux-based Node.js app, so the file here says start building this image by first grabbing the node:current-alpine image. Now, hmm, okay, this is actually a special container image with node tools preinstalled, and we're going to use it as the foundation, or the bottom layer, of our image to build everything else on top of. Now, a quick side note, right. Okay, this is not a full-blown Linux distro with a kernel and everything, right? It's more like maybe a set of file system constructs, like folders and device files and stuff. Because what happens, right, is every container, when it is running, uses the kernel of the host it's running on, so the container itself doesn't come packaged with its own kernel. Anyway, on top of a kernel, an operating system has config files and device files and other stuff, so this image here is a set of Alpine Linux constructs, plus, of course, a bunch of node stuff. Now, this is a getting started course, right? If you need more details on how containers share a host's kernel and all kinds of internal stuff like that, then Docker Deep Dive here is what you want. It covers all of that in glorious detail. Anyway, next up is just a bunch of metadata telling you who to hustle about the app. Ignore that. But then this line here is saying in this image make a new directory called usr/src/app. Then the next line is let's copy in our app code, and into that folder we just created, it literally is a set of instructions Docker just iterates through. Now, however, the period here says copy in all the files and subdirectories from wherever we run the build command. So on my machine here, I'm going to be running the build command in a minute from within the folder that's got all of the app code. We'll see it in a second. Next up, though, we set the working directory to where we just installed the app. This line installs the app dependencies listed in package.json, and then this entrypoint here starts the app, or it's the command to run each time a container will get started from this image. It's basically calling node with the name of the main app file. And that's it. That is how Docker is going to build the image. Start with the Linux image with node tools, add some metadata, create a directory for the app, copy in the app code, set the working directory, install dependencies, and set this as the command to start the app. Flipping magic! Now look, we're a getting started course, I don't know how many times I've said that ,so we're only wetting the appetite and painting the picture of what this is all about. We've plenty more courses to take you to the next level. But you know what? Stop waffling, Nigel. Let's build it! Now at this point, if you're wanting to follow along, you're going to need two things, Git and Docker. If you're on your laptop or whatever with Docker Desktop, then just Google how to install Git. It is properly easy. If you're following along on Play with Docker, ha, even better! Git's already installed. Anyway, this here is the repo, and I want to copy this. It's just the URL to the repo. Then from a Command Prompt, I go git clone, and I'll paste that URI in. Now look, if you're new to this, all that's done is it's made a copy of that repo with all of the source code onto my local computer. Now, actually, I want to switch into that directory, and here are all of the files. Exactly what we saw on GitHub, yeah? Okay, so the command to build an image is docker image build, or you can just go docker build for short. I am going to tag this as nigelpoulton and gsd for getting started with Docker. Then I'll finally say first container. Reckon that will do. Yeah. Now then, look, this bit here is my Docker Hub ID. So yours will be different. Now, you can use mine, and if you do, some of this stuff we're about to do will still work. But trust me, it's way better if you use your own. So again, if you haven't already got your own Docker Hub ID, maybe pause the video and head over to Docker Hub and get one. They're free, and they're pretty important if you're serious about working with Docker. Anyway, look, this bit is your Hub ID, then this bit is the name of the repository, and then this is the actual name of the image. Normally this last bit will be some sort of version string, but honestly, it can be anything, and this will work for

us. Well, then we say period. Now, the period is important. It's telling Docker that when we fire off this command, all the files it needs, especially the Dockerfile with the build instructions, are in the directory that I'm running the command from. And away that goes! And if we look closely, well, actually, I'm going to warp space time a bit here so that you see things happening as I explain them. Well, Docker is iterating through those instructions in the Dockerfile, the exact same instructions, yeah, So it is pulling the node alpine image, creating that directory for the app code, setting the working directory, installing all the dependencies, and then dumping out the image. Now, we don't see the entrypoint. command, as that just gets stored as metadata. But this, right here, is the image! So the app, and all dependencies, all wrapped up nicely, and here it is on my machine. How easy was that?

Hosting on a Registry

Alright, we've taken some app code and built it into a container image, but right now that image is landlocked on your local computer. And you know what, that's alright if all you want to do is run a container from it on that machine, but in the real world, you're going to want to host it somewhere where you can easily access it and use it from different environments, which, is where centralized repositories like Docker Hub come into play. Now, I've got to say, there are loads of container registries out there. I mean, for sure Docker Hub is the most popular, but Google's got one, GitHub got one, there genuinely are loads, and you can even host your own on your own private cloud or even on-premises. The point being, container registries are where we store container images so we can share them and access them from different environments. Anyway, when we built the image, we tagged it with this, and I think I said something like, it's best if you use your own Docker Hub ID instead of mine. Well, if you followed my advice and you did use your own, you'll be able to use docker image push to upload it to Docker Hub. So I'm going to go docker image push, and then it's just the name or the tag of the image. Now, if you try this with my Hub ID, it's going to fail because you don't have permission to push it up to my repositories. Actually, do you know what, even if you're pushing to your own repositories, you might need to do a docker login. Anyway, look, that's pushed, meaning if we look at Docker Hub, and again, this is my account so it'll be different for you, but it's this gsd one here. And this should be our image, first-ctr. And see how it's recognized it as a Linux app? Pretty cool. Anyway, look, that easy again. At this point, we've taken some regular app code, packaged the app and all of its dependencies as a tidy little image, and now we've lobbed that up to a centralized repository, just too easy. Well, do you know what, let's go and run it.

Running a Containerized App

Now then, if you've been following along you'll have an image on your local machine with the app and the dependencies inside of it. And you can totally use that local image and run a container from it, but I want to demonstrate Docker Hub so I'm going to delete that local copy. Though, before I do that, it's a good thought exercise to think of an image as being like a stopped container. And then, on the flip side, a container is basically a running image. So, if you know virtual machines at all, and I'm figuring most of you will, well, a VM template is basically a stopped VM. And then on the flip side, a VM is a running instance of a VM template. Kind of sort of, yeah? And if you're a developer the same kind of goes for classes in

object-oriented programming languages. So, an image would be like a class and then a running container would be like an object created from that class. Huh. Long story short, images are build-time constructs and containers are runtime constructs. Anyway, I am going to delete that local copy of the image. Give it a quick double-check. All right, we're all gone. Now, to run a container from it is docker container run. The -d flag here says to run this container in the background detached from my terminal. And this will make more sense in a minute. Then we give it a name. I'll call it web. You can call yours whatever you want. And then we'll do some port mapping. So, hang on a sec. This is basically saying map port 8000 on the Docker host, which for me is my laptop because I'm running on Docker Desktop. If you're in Play with Docker that'll be port 8000 in your Play-with-Docker instance. But then this other part here, 8080, is the port that the app is listening on in the container. So if we come and look at the code again real quick, I think I said to remember this earlier on, but look, the application itself is working on port 8080. Well, back here the port mapping is saying any traffic hitting your Docker host on 8000 is going to get sent to 8080 in the container and to the app, of course. Well, we've also got to tell it which image to use. Now, Docker is opinionated. That means in this instance if we don't stick a URL of a different registry in front of the image Docker is going to think we mean Docker Hub. Well, away that goes. So, the top line here says, well first off it looked for a local copy of the image. But I've deleted that, remember, so it didn't find one. So, it went to Docker Hub, pulled it, and it started a container from it. And this hideous long string here is the ID of that running container. Only, is it actually running? Looks like it is to me, this here being a short version of that container ID. And as well we can see the image it's based on. This is the app command that we documented, remember? And then whatever the port mappings are in a name. So, Docker says it's running, but you know what we can do even better than that. It's a web server mapped to port 8000 on your Docker host, which, if you're following along on Docker Desktop, is your local machine or your local host adaptor. If you're on Play with Docker you'll have a nice button with the port number on it that you can just click. And then I guess if you're on another service in the cloud, probably, all you'll need is the public IP or DNS of the cloud instance you're running on and then port 8000. And look, there we are, a running web server. Whoo. So, we have gone from bog-standard, uninteresting source code on GitHub and then used Docker to build an image, pushed to a registry, and then pulled from a registry and run as a container. Walk in the park. Well, let's see if we can go and stop and restart it.

Managing a Containerized App

So, we've got a web app running inside a container. We're not messing about, are we? Anyway, you know how the industry loves a good buzz word? Well, we call what we have got or what we've done a containerized app, and it's kind of like a super fast, lightweight virtual machine, meaning, we can stop and restart it, just like we can with a VM. So, funnily enough, docker container stop is the command to stop a container, and you can give it the name of the container or its id. I called mine web, so we'll go with that. Now, ah ha, yeah, it can take a few seconds while Docker gives the app running inside the container a chance to gracefully shut down. Basically, what it's done is it sent the app a SIGTERM signal, and then I think it's allowing something like 10 seconds of grace for the app to shut itself down, and if it doesn't shut itself down gracefully, it'll be terminated with a SIGKILL. Anyway, if we list containers again, but with the -a flag, we can see it's still listed, but it is showing as Exited, meaning, if we hit refresh in our browser here, okay, not responding. Well, yeah, we

just stopped it. Then, if we start it up again with docker container start this time, same name, of course, another quick refresh, and we're already back in business. So starting and stopping containers couldn't be easier. Well, guess what? Deleting one's pretty easy as well. So, I'll stop it first. Remember, give that a second, yeah, and then I'll delete it with docker container rm, and then the name of the container. Then, if we run that ls -a command again, not a trace, right? Meaning, obviously if we retry the start command from a second ago, nah, no joy. The container is literally wiped from off the face off the earth. Now real quick, when we run that container, we use the -d flag to run it detached from the terminal, and that's fine for containers that are designed to run in the background, like normal web servers and the likes. But, you can run containers in the foreground attached to your terminal. So, a really simple example might be docker container run again, only this time -it for interactive and terminal. Call it whatever you want again, and we'll base this one on the base alpine image and run sh as the main app to run inside the container. Now, another example might be this: if you're running Docker on Windows Server or even Docker Desktop on Windows, if you are in Windows container mode. Anyway, look, if you look closely, you'll see how my shell prompt changed. That's because my shell is now attached directly inside that running container, meaning, any time I run a command here, it's actually running inside the container, I mean, that's pretty cool, right? We are inside of a container running commands from there. However, if I type exit, that will drop me back to my terminal on my Mac. But it'll also kill the container because I'm effectively killing the shell process that I said was going to be the container's main process, and when you kill the main process in a container, PID 1, if you know your Linux, then the container basically throws its arms up and says, I've got no more work to do. I might as well terminate. Well, I don't want that actually, so, instead I will type ctrl+p+q, and don't ask me why it's p and q, but we can see that has dropped me back to my local terminal, and if we list the containers, yeah, look, it is still running, so, ctrl+p+q is a graceful way to leave a container without burning it to the ground as you leave, and then, this time to terminate it, I'm going to do it with the -f flag. This basically says to Docker, look, I know that the container's running and you don't normally like to delete running containers, but trust me, just annihilate this one. Magic. Tell you what, though, it's been a lot, so let's do a quick recap.

Recap

So, we know the crack by now. We started out with regular app code, and that's important because Docker doesn't care. It is language agnostic. So you can start using Docker right now without having to learn any new languages. Now, you might have to learn some new tools, but Dan's going to have you covered with some of those in the next course on the Docker for Developers learning path. Anyway, look, we took that code and we used Docker to build it into an image, push it to a registry, and run it as a container. We even stopped it, restarted it, and deleted it. Marvelous! But why, Nigel, pray tell, should I care? Well, and I know a bunch of you get this already, I get it, right, but not everybody might get it. Well, look, on the operations front, containers are the future and they're kind of virtualization 2.0. So, we know that hypervisors virtualize hardware, like virtual CPUs, virtual RAM, virtual networks, all of that jazz, yeah. Well, containers do a similar thing, only a bit higher up in the stack. They virtualize operating systems. So each container is basically a virtual operating system, so it has its own process tree, its own root file system, its own eth0, and all the rest. So, kind of like how every VM on a host shares the same hardware, every container on a host shares the

same OS kernel. And because there's only a single OS kernel in the container model, containers are smaller, faster, and more lightweight than virtual machines, meaning more applications per square foot of infrastructure. Again, that's because containers are just app code and dependencies. Like, if you run, I don't know, 50 containers on a host, let's say, they all share a single OS kernel, hence, smaller, faster, and ultimately, more applications per host. Plus, there's a whole bunch of new tools to manage containers and all of that new, shiny, underlying infrastructure. Plus as well, there's a bunch of new paradigms when it comes to security and shared kernels and stuff, so you really want to know this. So, that's a good part of why you need to care as an operations or an infrastructure professional. Now, on the developer front, you care because you can develop apps on your laptop with your favorite languages and tools, and Dan's going to go into more detail about that in the next course on the developer path. But you develop locally in the security that if it works on your laptop, it is going to work in production, all because of the way Docker builds your apps into an image. Remember, it packages the app, of course, plus all dependencies, meaning, sorry if I'm overstressing this, right, I feel like a bit of a geek sometimes, but the thing is, because all of the dependencies are packaged with the app, gone are the days of where something works on your laptop, but it bombs out in production because maybe you're running different libraries or whatever. They're not in the container model because the libraries get shipped with the app. As well, though, if you're a developer, the tools, the portability of code, unloads more things and making containers more and more the pattern of choice for developing modern apps. Oh, you know what, though? I reckon that'll do for a recap. Just let me finish by saying one last time it was gloriously easy to get that app from source code to running in a container, and who in their right mind doesn't want more of that? Anyway, we're not done yet. So far, yeah, we've seen how to containerize a ridiculously simple app using the Docker CLI and run it on a standalone Docker host. Well, coming up next, we'll look at how to easily build a more resilient infrastructure to run your apps on, and we'll see how to define and document more complex apps in declarative YAML files that just make it so much easier to deploy and manage them. Oh, and of course, you'll see it in action and get your chance to get hands on. See you there.

Microservices and the Real World

Module Overview

Okay, your time and attention are valuable, and I am massively grateful for them, so let's not waste them. This is what we're about to cover. We'll kick off, actually, by busting some jargon, so we'll be saying what we mean when we use terms like cloud-native and microservices. Now, this is important because, well, they're real things, and they impact both developers and operations professionals, but, as well, I don't want to make assumptions that everybody automatically knows what they mean. Well, look, when we're done with that, we'll get our hands on with a crazy cool developer tool called Docker Compose. Now, you're going to want to watch this even if you're in Ops, because we'll be showing multi-container apps and we'll be introducing things like microservices and declarative configuration. Now, of course, those might be buzzwords right now, believe me, I know, but as we crack on, I promise we'll explain them all. Anyway, look, then we'll take a look at something called Docker Swarm. Now, this is a simple, but a powerful alternative to

Kubernetes, and we'll see what it brings to the operations and implementation side of things with clustering, high availability, and security. Well, once we've got a swarm built, we'll look at what a Docker service is, and at this point, it'll be important that you understand microservices architectures, so don't skip this lesson up here. After that, we'll wrap things up by looking at Stacks, which are multi-container apps like Compose that we've already covered, but they've got a more production-like implementation or a more production-like feel to them. And then, when we're done with that, seriously, you'll know exactly what Docker offers on both the operations and developer front, and, obviously, you'll be properly primed and ready to explore things even further. Now, I get it that this might look a lot, and I also get it that I'm throwing around buzzwords like I think maybe it makes me sound clever or cool, and I can assure you I do not think that. But, what you'll learn here will set you up, well, nicely as a developer for attacking things like multi-container microservices apps done declaratively. Heck, I'm going to have to wash my mouth out, so many buzzwords, but remember, we're going to explain them all. Also, though, on the operations front, you'll get a solid grasp of the new way of developing, as well as managing, all of these fancy cloud-native, whatever you want to call them, modern apps. And, of course, you'll walk away with a decent idea of how to start doing it all on a highly available, secure by default, clustering platform. And buzzwords aside, between you and me, honestly, what we're about to see is properly game changing. But you know what, enough of me waffling, come on.

Cloud-native Microservices

Okay, so I want to keep this as brief as possible, but I also want to make sure you get the picture. Now, okay, before I go any further, the definitions you're about to hear are according to Nigel. Now, in no way am I trying to say they're canonical, and I totally get that people can be passionate about their own definitions, so let me be clear, in no way am I trying to say that my definition is better than yours, though, obviously it is. No, I'm just kidding. The point is, the definitions here are broadly correct, but they are tailored a bit towards the content that we're covering in this course. Anyway, look, we'll define microservices first, and whenever I do this, I usually find it best to start out with a legacy non-microservices app. So this is a picture of an app, and the point is, while it's a single application in binary, it's actually made up of lots of small features. Now, of course, I'm massively oversimplifying, but the point is the web frontend, the data store, reporting, logging, the whole picnic is part of a single app, so it is deployed and managed as a large single unit, meaning, if you want to maybe patch or push a feature update or something to, say, the reporting service, well, I'm sorry to break it to you, you are patching the entire app, and that can sometimes mean taking the whole thing down while the patch is applied. Yeah, the whole thing, and, of course, on a weekend where you'd rather be at home or fishing or golfing or pretty much anywhere other than the office. As well, though, in this kind of design, if you need to scale one aspect, maybe the reporting, well, you can't. You basically scale the whole thing or none of it. I'm sure, you know, like move it to a more powerful cluster or something, lock, stock, the whole thing. Now, again, I'm over simplifying of course, but big and clunky was the mantra. Well, microservices takes that same application experience, and it breaks each feature out into its own smaller, discrete service, hence the term, microservices. Now, in this model, each microservice usually is coded independently and often by different teams, and that's the microservices design pattern. Take the different features of an app and break them all out and code them independently. This then leads to cloud-native features, so things like patching,

dynamic updating, dynamic scaling, all of that jazz, but against each individual microservice independent of the rest. So, do whatever you need to do to one part of the app without touching and potentially breaking something else or everything else, right? Now, again, I'm oversimplifying, but that's the gist. Now, one last thing, the term cloud-native absolutely does not mean that it will only run in the cloud, not at all. I mean, for sure it will run in the public cloud, but also your private or hybrid cloud, including on-premises, basically anywhere you've got something like Docker or Kubernetes. Aah. Well, with that in the bag, let's crack on with the examples.

Multi-container Apps with Docker Compose

Okay, so far, we've seen how to take the simplest of apps and then use a handful of Docker commands to go from code to app, and that is the bizzo, don't get me wrong, but it's a bit like 1990s? I don't know. By that I mean it's not exactly scalable, and it is a bit opaque. Like with all of those Docker commands we've used, it's never really been obvious what the app does. Plus, it's not a great way of showing how different app components connect. Well, a better way is the declarative way, which, listen, is basically a fancy word for saying you define everything you want in a config file in how it all connects, and then you just give that config file to Docker and let Docker do the rest. So, this file here, among other things, it defines a multi-container, or a microservices app with a web front-end and a Redis backend. And, honestly, we can throw this file at Docker and Docker will, well, it will build or pull any images it needs, it will create any networks and volumes and secrets and the likes, and then it will start the containers. And I think you're going to love it. Well, do you know what? To do this, we actually need an extra tool called docker-compose. However, if you're running Docker Desktop on Mac or Windows, magic! You've already got Compose! And if you're on Play with Docker, ha, it's there as well! However, if you're on Docker on any other platform, you're probably going to need to manually install it. But you know what? That's dead easy. Just Google how to install Docker Compose. I promise it's a piece of cake. Well, once you've got it, you're going to need to be in the multi-container folder of the course's GitHub repo. And then, you know what, actually, let's look through some of the files first. Now, I'm going to do this from GitHub because it's probably a bit easier to read. Well, this file here is the main app file. It's a Python Flask app, or actually, it's a Python Flask app that talks to a Redis cache, and basically, every time you hit the web page, it increments a counter in the cache. Anyway, look, that's the app code. Well, this file here lists the dependencies, and we know what the Dockerfile here does. So, app code here, requirements here, and instructions on how to build any container images here. But this Compose file, this is new, and it's pretty exciting, actually. So look, it is defining two application microservices. This one called web-fe is the Python Flask app. This will call that Dockerfile to build an image and then set this as the app to run when a container starts. It's saying map port 5000 on the container to 5000 on the Docker host, it's attaching to this network, and its mounting this volume. Now, of course, we could do all of this stuff by hand on the command-line, but I mean, come on, it's so much cleaner to document it all here and then leave the commands and the hard work to Docker. However, there is a second service here called Redis, but this one just pulls a stock image from Docker Hub and it attaches to the same network. And then, of course, these bits here tell Docker to create a network and a volume. Now, the power of this declarative approach, that's what this is here, by the way, instead of maybe 10 Docker-specific commands pulled together with bits of string in a script, we define our desired state in a file

like this. So, I want a web front-end container built from a Dockerfile I'm giving you, listening on port 5000 on the counter-net network, and please mount the counter-vol volume. Oh, and I also want a Redis service on the same network. And do you know what? We let Docker care about how to, well, actually build and pull any images, build and attach to networks all the volume stuff, and how to start the containers. Like, I can't be bothered remembering how to do all of that; let's just have Docker do it. So, and there's more to this, right, but from a developer point of view, this is way better than remembering a ton of Docker commands. Plus, it's actually a great way to document and keep track of your multi-container apps. Well, then, from an operations perspective, it's pretty much living documentation, and honestly, I know that sounds super cheesy, but while this is the source of truth for Docker when deploying the app, it's also pretty solid documentation for you in operations, like you know which images are being used in networks and volumes and what ports are exposed. It's pretty sweet. Anyway, look, from within the multi-container directory containing all of the files, you'll literally go docker-compose up. You know what? I'll add the -d flag here just to run it detached in the background. Now, the command works because Compose expects its config file to be called docker-compose,yaml I mean, you can call it something different if you want, but if you do, you need to pass it the file name as a separate argument. But you know what? That's looking good. So if we list images here, yeah, there's the Redis image, this one is the Python image pulled to build the actual app image here. Then if we list containers as well, ugh, don't like that. And let's see if I can make that a bit better. Oh, hello. You know what, that'll do. I am never fitting all of that on one line. Anyway, this one is the web front-end one, and this one the Redis back-end. You know what? Look, you can list networks and volumes, if you like. The point being, loads of Docker constructs built, but from running just a single compose up command referencing that moderately easy-to-read YAML file. Anyway, look, we're on port 5000, meaning If you're on Docker Desktop, you can browse to localhost on 5000, and there's the app. I suppose I better give that a few clicks. Kind of rude not to. Whew, that took a while. No, just kidding. Look, if you're following along with Play with Docker, you'll have a button in the UI with the port number on it. So just click that and you'll see the app, and remember, give it loads of clicks, yeah? If you're running Docker somewhere else, it's probably going to be some combination of a host IP address plus port 5000. But that's it. I mean, we just used docker-compose to deploy a multi-container app. And if you like your buzzwords, that is a microservices app with two services, all from a declarative config file and deployed to a single Docker host. Well, do you know what, to bring the app down, it is just docker-compose down. Give it a second. All right. Now look, of course there's loads more; there always is. But right now, at this point, you know what docker-compose is, plus you know a bit about declarative configuration. Only, I don't know, it's all a bit, I don't know, laptop-y, if that makes sense? Actually, maybe hacky is a better term. I mean, look, it's cracking for local development, but what about something that looks and smells a bit more like production? Well, funny you should ask, because that is next on the agenda.

Taking Things to the Next Level with Docker Swarm

Right then, Docker has this mode called swarm mode that lets you cluster multiple Docker hosts into a secure, highly available cluster. Now, we've got courses that go into way more depth than we're about to, but what we're about to cover here will give you enough to get a swarm up and running and at least wrap your head around what it means to you as either an Ops professional or a developer. Now, on the theory side, the cluster comprises managers and

workers, and actually, we call the cluster a swarm. So a swarm is a cluster of one or more manager nodes and some worker nodes. The managers host the control plane features, so things like scheduling and persisting the state of the cluster and the apps that it's hosting. So, in production, it is really important that the managers are highly available. Now, the recommendation is to have an odd number, usually three or five. The odd number is to avoid a split-brain condition where there's maybe a network issue, and you end up with an equal number of managers on both sides of the split, because the issue in that situation is that neither side knows if they have a majority, and then updates to the cluster are frozen. Obviously, if you started out with an odd number of managers, then there's less chance of a split brain. Like here, this side of the split knows that there were five, it can't communicate with two anymore, so it knows it has a majority, and it keeps the cluster open for updates. Obviously, then, these over on the other side know that they don't have a majority, so they won't make updates to the config. Anyway, managers and worker nodes can be whatever you like, so on-prem or in the Cloud, VMs, physicals, it doesn't matter. In fact, all that does matter is that they have Docker installed and can communicate over reliable networks. Now, if you are running with Docker Desktop, you can still run in swarm mode, you're just limited to a single manager node that does everything, so no HA or anything like that, which, might sound kind of like, well, what's the point Nigel? Well, as well as clustering nodes for high availability and the likes, swarm mode also unlocks additional Docker features, most notably, services. Now, if you just rewind your brain a bit to where we defined what a microservice is, well, a Docker service, and we'll see one in a second, but a Docker service maps directly to an application microservice, but look, we'll get to that in a second. For now, let's build a quick swarm. In fact, I'm going to build this one here with three managers and two workers. Yours can be different, but the process is the same. In fact, if you're following along on Docker Desktop, you're just going to do the first step. Anyway, from one of your Docker nodes, you need to initialize the swarm. This will make the node that you run the command on, the first manager node in the swarm or the cluster. Now, like I said, if you're running on Docker Desktop, this is all you need. You can hit Return and you'll get a single-node swarm. However, on Play with Docker and most other places, you're going to need to specify this flag here. What it's doing is telling Docker which of the host IPs to use for the cluster communication. In Play with Docker, you can use the node's 192.168 address. And if you're in a public cloud like, I don't know, AWS or whatever, you should just use one of the instance's private IPs. Anyway, look, we have got a swarm initialized and this node is the first manager. But I want three for high availability, so I'll run this command here to get the command to securely join a new manager to the swarm. Now, look, obviously it includes a join --token, so keep this safe, but I'll have that, and I'll run it over here on this node that I want to make the next manager. Marvelous! And the same again for the third manager. Bingo, right. At this point, we should have three managers. Sweet! So what are we looking at? Well, yes, three managers, and look, node 1 is acting as the leader. The others are followers, but they can step in and become leaders If node 1 ever goes down. Anyway, look, adding workers is pretty much the same. So if I jump back to node 1 here and then rerun that join --token command again, but this time for workers, grab that, and then run it on the nodes that I want to make workers. Now, once that's done, run this again, three managers and two workers. So that's us with a five-node swarm running three highly available managers and a couple of workers. And, of course, look, in the real world you'll distribute all of that across whatever your infrastructure failure domains and the likes are, but for us, we're ready to see what it looks like deploying an app to a swarm.

Microservices and Docker Services

Okay, so with a swarm up and running, you unlock a few pretty cool additional Docker features, one of which, as we said, is the service object, and another is the docker stack command. Well, we'll look at service objects first. And one last time, a Docker service object maps back to an individual service in a microservices app. So, do you know what, real quick on the old PowerPoint here. Each one of these application microservices, each one can be implemented through its own Docker service, and then you deploy it and manipulate it via the Docker service object. Yeah, well, look, if trying to figure out that is hurting your head a bit, don't stress, I'm pretty sure you'll understand it when we actually do it. But right now, for example, if you need to scale up the microservice, you do that by altering the docker service associated with it. Now, look, just to be clear, there's 1, 2, 3, 4, 5, 6, 7, 8 microservices here, so that would be 8 Docker services, so one for the web stuff, one for the reporting, another for logging, you get it, yeah? Anyway, look, we're going to take our first look at Docker services via the imperative method where we use the Docker command line to manage everything. Once we're cool with that, we'll see how to do it all declaratively with the docker stack command and a declarative YAML file. Hoo, okay, so the command to imperatively create a Docker service is docker service create. Now, this is only available in swarm mode, so if you've got Docker and you've not initialized a swarm, it ain't going to work. Anyway, look, I will call this particular service, web, then, as I'm using the same app as before, I'm going to map 8080 this time to 8080 in the container. And then this --replicas here lets us say, funnily enough, how many replicas of the container we want, and I reckon 3 for now. Well, like I just said, we will base it on the same image we used before. And you're totally welcome to use this image if you're following along, but you know what, if you were following along and you've built your own, by all means, use that. Anyway, off that goes and Docker is spinning up three identical replicas or containers all running from that exact same image. Now, we can verify stuff with the Docker command line. So listing the service shows us that three out of the three replicas are up, and then the usual image and port stuff. Now, the right way to manipulate the service is with the docker service command, but, as we can see here, we can go in, I don't know, sort of through the side door with some of the Docker commands that we're already familiar with. So look there, we can see three containers, magic! But remember, a service replica is a container, so we asked for three service replicas, and we've got three containers. Ha ha! However, if you're running this kind of command on a multi-node swarm, then docker container ls is only going to show you the containers running on the local node. So if you're on like a five-node swarm, you're not going to see all three with this command. Like I said before, the better command is docker service ps. Well, for starters, you can run this on any manager and see all replicas. Plus, you get to see the node that each replica is running on. Now, okay, I didn't tell you this earlier on, I've actually flipped back to a single-node swarm in Docker Desktop, so I've only got a single-node cluster. But, if you built a proper swarm in, like, Play with Docker or somewhere else, you'll see replicas load balanced across all nodes in the cluster. Either way, though, do you know what, three identical containers, and all up and running. But, I always say, believing what Docker tells us is one thing, seeing it with our own eyes is another. So, like I just said, I'm on Docker Desktop, so it'll be localhost and 8080 this time. And there's that web server again, only this time, take note of the container that serviced the request here. If we hit Refresh a few times, look, we see that cycle through the three container replicas. You see that, yeah? Now, these names match back to the container names on the command line here. And this is magic, right? Only, do you know what, it is not even the half of it. So, we can

scale the number of replicas with docker service scale, and then just the name of the service, and however many we want, and off that goes. And if we check, ha, that looks like ten to me, and actually, see how some of them are newer than the others. These will be the new ones. But then, let's say we take an ax to it, maybe the top three. Give me a second, ha, ha, ha. Okay, so this will go in through that side door that I mentioned before and destroy three containers. Uh-oh, oh, okay, yeah. I need to force that. Okay, they should be gone. Only, if I give this, I don't know, maybe 2 or 3 seconds, and then we give it another try, check that out, we're back up to 10, and actually 3 of them lock in very new. So, what happened there? Well, Docker knows we'd asked for 10, so it recorded that as our desired state. But then we went in through the side door, so not using the docker service command, and we blew three of them away, basically a crude way of simulating failures. Anyway, Docker was like, well, look, I'm supposed to have 10, but for, well, whatever reason, I've only got 7. So, it fired 3 new ones up and took us back to our desired state of 10. And all of that cleverness is managed via a reconciliation loop that is constantly watching the state of the cluster and comparing the observed state with our desired state, with the goal of the loop being that observed state should always match our desired state. Now, look, if I'd gone in through the front door and changed the state with the docker service command like I did with the docker service scale earlier on, then it's all good. Docker knows that any of those changes are intentional. But if we use other Docker commands or, of course, if something actually crashes or a node fails and we drop from our desired state of 10, then Docker knows that it's not intentional and it tries to fix it. But I'm waffling, look, that's the imperative way, and, yeah, it's okay, but a much better way is the declarative way with the docker stack command and a declarative YAML file. Let's see it.

Multi-container Apps with Docker Stacks

Okay, so let's do all that again, yeah? Hey, it wasn't that bad! Anyway, look, this time we'll do it all again using a declarative YAML file. Because, like I said before, documenting an app configuration in a YAML file, it just helps on so many levels. Well, as a quick reminder, the most obvious of which, or at least one of the most obvious, is that you actually have your app documented somewhere. And from an ops perspective, that is like sit down and breathe into a paper bag to contain your excitement. But as well, in the dev world, it makes it so much easier for you to pick up an app later on down the line and easily refresh your memory. Or, do you know what, even bring someone else on the team or even a new hire up to speed. Just give them the YAML, and it's pretty easy to figure out the anatomy of the app. Whew! Anyway, first things first. Let's clean up any running services left over from the last lesson. So, I'm going to go in through the front door, yeah. That'll be docker service rm, and then the name of the service. And we'll do a quick verify, and then, do you know what? Why not? A double check with this. Give it a sec, actually. Okay, we're good to go. So, here in the swarm-stack folder of the GitHub repo, we've got another app, which, do you know what, if I'm perfectly honest, is almost identical to the one that we use with docker-compose. In fact, do you know what if we open up the compose file, yeah, it's got a web front-end and Redis cache. So, the app that counted the number of page hits, only this time it also returns the name of the container that served the request. Now, then, because we're running this app in Docker swarm mode, we call the app a stack. So this file describing the app with two services is describing the stack. Now, out of interest here, stacks on a swarm do not support building images on the fly. So, in the compose example earlier, this web front-end image was built on

the fly using a Docker file. Well, stacks can't do that. I guess they're more of a production tool where you're probably not going to be building at deploy time. Anyway, this time, the web front-end image needs to be pre-created. Now, of course, in the real world, it'll have been through all different tests and the likes. But also, it'll need storing in a registry so that all of the nodes in the cluster can access and pull it. We'll come to that in a second, actually. Now as well, this block here, deploy.replicas equals 10, this is saying give us 10 replicas of this service. So, similar to what we did imperatively a minute ago, yeah. But aside from those bits, it is pretty much a compose file like we saw before. So let's go and deploy it. Now, we said the image for the app needs to be pre-built, so we'd better do that first. Obviously, we've got the Docker file here with the build instructions, so we go docker image build, we'll tag it as, hmm, what did I call it? Oh yeah, well, you know what, might as well copy that. There we go. And don't forget the period. Remember? That says use the Docker file and the app stuff in the current directory, and off it goes. Yeah, go on. I will cast aside the laws of physics again to speed that up. Cool. Now, as I'm on Docker desktop, I could crack on at this point is I've got the image locally in only one node. But if you are running a multi-node cluster like you will in production, you'll need to push that to a registry so that every node can access it. Give that a second. Okay, so look, with the image built and the registry, we go docker stack deploy, the -c flag to tell it we're deploying from a compose file, and then we'll call the stack now counter. Now, that's the name of the app. We'll see it in a second. Okay, look, we can see it's built the network, then it's deployed the two microservices. Now, the docker stack command is pretty versatile, and it lets you list and introspect running stacks. Obviously, we've only got one running, but we can see that both of the services in the stack are up. You can see more detail on each services in the stack, and you can see each container. Now the name of the service or the microservice as defined in the YAML file is embedded as part of the container name. Check. But you know what, let's see if it is actually running. Oh, actually look, this is on port 5000. I just mapped it that way in the YAML file. But localhost and 5000, and there we are. And you know what? I don't know, I might make this look a bit fancier in the future, so don't be surprised if it looks different when you're deploying it. But the important thing is, right, that if we hit refresh, yep, the counter increments, but also the container ID here changes as well. Now that's because we've got 10 replicas of the web container running and requests are being balanced across them all. But also, if you scale up and down, your requests get automatically balanced against any new containers added. Though, I must point out, this is layer three load balancing, and it's pretty crude. It's not application or well layer 7 stuff. Well, anyway, look, one last thing, a recommended way to increase and decrease the number of replicas, or, to be honest, make any changes to the stack, is to crack open the same application YAML file, which in the real world, I am really hoping you're going to be keeping in a version control system. Anyway, crack that open, and you make your desired changes. So, we'll change the number of replicas, give it a save, and then just rerun that docker stack deploy command, and Docker takes care of realizing that this is an update to an existing stack, and it goes about updating the relevant bits. And the beauty of doing it this way, so checking out the config file, recording your updates in there, and then using that updated file to change the state of the app on the cluster, the beauty of that is that your YAML file is always up to date and remains that source of truth for both developers and operations. And on that note, folks, oh, actually, no. Let's just tear it all down first with a docker stack rm, I think we called it counter, yeah. And this time for sure we are done with the demos .But stick around for maybe, I don't know, maybe 2 more minutes while I list out some of the courses I recommend you take next as you crack on adding more and more vital Docker skills to your resume. See you there.

What's Next?

What's Next?

Wow! We're all done. Well, first off, let me congratulate you on making it to the end. An hour and a bit of listening to me? Ha! That is no easy ride. No, but seriously, congrats on completing the course. I am genuinely excited for you. I mean, you've got the fundamentals of Docker and containers now, and you're more than ready to press ahead with further mastery, which, on that note, you're in luck, because we are living in, what I see at least, as a golden age of technology learning. I mean here at Pluralsight, we've just got so much content for you. Like we've got the developer learning path and an operations learning path, and I highly recommend both, and I'm hoping you'll take courses from them. And remember, we're building them out all of the time, but we've also got standalone courses that I mentioned earlier, and couple of highlights are Dan's Docker for Web Developers, and my own Docker Deep Dive; well, if you can stand any more time with me, that is. But the thing is, these are just the highlights. Have a proper look, and I promise, you won't be disappointed. And as well as that, though, Kubernetes is very much in the mix with containers. And you know what? We've got you well and truly covered with that here at Pluralsight as well. I've got a Getting Started with Kubernetes course that I suppose is very much like this, only it does actually go a bit further and into a bit more depth, but as well, we've obviously got a ton of learning paths for Kubernetes as well. Now away from Pluralsight, but sticking to Docker, we've introduced you to a great tool here called Docker Desktop. It is a properly good tool for both learning and working with both Docker and Kubernetes, and I highly recommend you install it and just get your hands as dirty as possible. And I reckon on that note, I'm Nigel Poulton, and I don't know about you, but I have massively enjoyed sharing this course with you. So feel free to connect and reach out, and look out for your local meet-ups and, well, other industry events. I mean, I used to highlight container-specific events, but you know what? These days, just about every major tech conference has a strong container focus anyway, so yeah, look, thanks for your time, and keep in touch, and let me know how you like the course and how your container journey is progressing. Ciao! Ciao!