Course Overview

Course Overview

Hi, my name is Wes Higbee. Welcome to my course, Getting Started with Docker Swarm. I just love how easy it is to run about any application I can think of with Docker, pull an image, spin up a container, and off I go. At some point, though, we'll run out of resources on a given machine, and wouldn't it be nice if we could use the same workflow to spin up applications across an entire cluster of machines? In this course, we're going to see how Docker Swarm makes this possible. We'll start by creating a cluster of nodes. We'll look at deploying services or applications onto those nodes. We'll take a look at stacks to help us manage not just the services, but also networks, volumes, configurations, all the resources necessary to run an application. We'll look at how we update our application with Docker Swarm. We'll even take a look at one of the newest features, jobs. By the end of this course, you'll be prepared to provision a cluster and deploy complex applications to it. Coming into this course, I'll assume that you have familiarity with a standalone Docker node using the Docker CLI and perhaps even Docker Compose. All right, let's get started.

Joining Nodes to Form a Docker Swarm Cluster

From Standalone to Multi-node

Docker is one of my favorite tools ever. And a big part of that is because I can remember the history where we had a single machine, sometimes a developer machine, other times a server, we've got our hardware and our operating system and we have processes running on that machine, and I can remember the pain of deploying updates and dealing with situations like more than one application needed network access on the same port, unless maybe they were configured to use a different port, and that was something I had to worry about. Maybe they could be, maybe they couldn't be. Maybe I knew how to do it, maybe I didn't and had to spend some time figuring it out. Or, maybe two other applications need access to a shared library and one of them writes over the top of the others and so that other application dies then, so we have to figure out this problem, then try and isolate the file systems that the two applications are using, give them their own folders, keep them out of each other's business and hope that these shared libraries or frameworks don't need to live in a specific place. If, for example, we need different versions of them, maybe different versions of a Java runtime or an older .NET runtime that couldn't live side by side. And of course, containers helped us fix this problem by taking a single machine and slicing up the resources such that it looked like we have individual machines for each application if we want or each set of applications that are tightly related. It's not actually a virtual machine, but it's a layer of virtualization that looks almost the same by isolating what a process has access to. You can isolate the network cards so that each container, it appears as if it has its own network card with its own IP address potentially, even on a public network, no port conflicts in that case. Or, those two applications with the shared libraries, each of them can have their own copy, they can live in the exact same location on disk, and no problem because each application has its own file system, or it looks like it has its own file system, its own mounts. Of course, there are other benefits as well, i's not just the isolation. I think a big reason why Docker was so successful is because of this concept of a registry, Docker Hub, that distributes prebuilt images that contain more or less the file system for any given application. And what really made Docker take off was the fact that a lot of people got involved and you could find almost any application as far as server ‑ side programming was concerned, any application out on Docker Hub as an official image that you could trust, applications like MongoDB or NGINX or MySQL, or even something like Jenkins, both the server and the agent in that case. The images are available. They're officially provided by either Docker and the team that developed them or the team that developed the software itself. And so not only did we give each container its own file system, but those file systems are readily made and available for us. Now if you're like me, it didn't take you long to go from wondering what Docker is to finding how you can take advantage of it in your own development environment because that's one of the best places to start when it comes to learning about Docker. There are benefits to be reaped without even needing to risk any sort of production environment or even a testing environment. There are years worth of benefits just in development alone or your personal workstation, even just a computer at home that you use. For example, I found myself running TeamCity to perform demonstrations. When a new version would come out, I'd just pull down the new images, they'd be up and running, and off I go with a demonstration of the features. And the night before I could poke around, see what's changed, if there were beta builds I could pull those down as well. I can even run agents as if they were separate nodes simulating clusters. I could do demonstrations of many, many nodes involved. So I could show off cluster ‑ oriented features, which is very hard to do in a traditional demo. And then maybe behind the scenes, or at least paused for a demonstration of TeamCity, I might also have a .NET application that I'm developing, and that .NET application might use a Mongo database or maybe Maria DB, MySQL, even Microsoft SQL Server is available now in an image, and not just for Windows, it's available with Linux as well. In fact, I would say that Docker is one of the reasons that pulled or pushed Microsoft in the direction of becoming much more Linux friendly. And with all these web applications on my machine, I might want some sort of proxy involved so I can front requests from the outside world if I have a demonstration environment or even a production environment and open this up to my local network or otherwise, and I might use something besides NGINX. There are a ton of alternatives out on Docker Hub. At the end of the day, what we run is less relevant than the fact that we have a lot of containers, and thankfully we have big, beefy machines today so we can run a lot of them. But that doesn't mean that at some point the resources won't become limited on that machine. As we pull more and more containers down, it's easy to run out of disk space, for example, just storing all those images, and thankfully those images are stored in a way to reduce disk space using layering. However, the easier that you can spin up different versions of any application also makes it tempting to keep lots of copies around. And of course we can grow our machines bigger, and this will work to a point and probably will work for a long time for a development machine, but even for development, and especially for production and testing environments where you have a lot of load, a single machine is just not going to cut it. And beyond just reasons of resources, also for reasons of reliability, failover, availability, even the ability to have access to different platforms because the ideas that made it possible to containerize an application and distribute it with an image, well, wouldn't that be a really cool thing if we could use it beyond just a single machine? Of course, we could install Docker on multiple machines, and if you're like me at first you probably tried this where you would remote into each of these machines and control what's running on them. It was a rather feasible approach at the start, but as your needs grow larger for resources and availability and failover, jumping onto each box is not so much fun, even with the latest Docker context features that make it pretty easy to switch context to point out different Docker installations. The truth is it would be nice if we could think of this as one giant machine, even if there are many involved. I mean, wouldn't it be nice if the number of machines was just completely irrelevant as far as you were concerned? And, of course, heavy lifting has to be done behind the scenes to make that possible, but that would make it really cool for an end user. And perhaps what's most valuable about Docker is making things easy for the people that use it, whether you're a developer, a tester, somebody that supports and works an application, the tools they've produced have always done a really good job of sweeping under the rug the complexities, how things happen, so that the what is very obvious and it's pretty easy to work with once you what's possible.

Initializing a Single Node Swarm with Docker Swarm Init

When it comes to clustering with Docker Swarm, I've found that it's helpful, and this would apply to other clustering tools as well, it's helpful to split out two concepts. We have our container that we've simplified, and then we also have the node, regardless of what's running on it for an operating system, regardless of what hardware it has. So we want to keep these two separate concepts isolated to a degree. Each of these parts, there are oversimplifications, but each of them represents what is a separate management lifecycle. We have to go about the process of creating nodes, getting Docker installed on them, and then we can go about the process of joining these nodes together so that we can start to take advantage of the features of Docker Swarm that allow us to think of everything as one big machine, one big node that has all of our containers, or whatever resources we'd like, running on top of it. So, why don't we do that? Let's go ahead and go about the process of creating our first swarm. So let's backup here a moment. We've got our three separate conceptual nodes. And one of the easiest ways to get started is actually just forget about the other two nodes and focus on a single node cluster. After all, you have to start with one node before other nodes can join to it. So if we create a cluster of one node, we can work with a lot of the interfaces to Docker Swarm, specifically the CLI commands, without the complexity of multiple machines involved. So for this scenario, we just need to find a single node that has Docker installed on it. Where might that be for most people that have some familiarity with Docker already? Given the prevalence of Windows and Mac OS, the Docker Desktop builds for each of these operating systems is a great place to start. After installing Docker, you can simply hop over to the command line and we can run a command called docker info. Underneath the server side, there's a Swarm section and it's labeled as inactive. You may have heard of Docker Swarm referred to as Docker Swarm mode, and that's because it's like a toggle that you can turn on and off. All you have to do is run the docker swarm command, and I won't actually run the sub ‑ subcommand necessary to do this yet, I want to show you there are a list of commands available, and inside of that list you can see there is an init command. This will help us initialize a single node swarm. We can also choose to join another swarm, but the technologies inside of Docker Desktop for Mac and Windows, how they fuse with your operating system to provide a transparent experience, doesn't always make it so easy to network with other Docker nodes and form a multi ‑ node cluster. I would recommend sticking with a single node, and in that case we need to run docker swarm and then init to create our cluster. And within a split second, we are done. That's all it takes to flip that toggle. Question. What do you think is going to show now if I run the docker info command again? Now when we scroll up here, we have a lot of information underneath the Swarm section. In this case, a single node swarm with a single manager node. And most telling, we have active instead of inactive, telling us that toggle is turned on.

Cloning the Course Repo and Explaining the Multi-machine Vagrantfile

All right, for our next example I want to take a look at a multi ‑ node setup using Vagrant to spin up VMs locally. And to do that I have some files prepared for this out in this course repository, there will be a link in the exercise files. Grab a copy of this repository and then hop over to wherever you'd like this to be cloned down at. I, for example, will put it into swarm ‑ gs right in my home directory. Keep the pathing simple. Change into that directory and list out the files. And inside of there there's a labs folder. Inside of the labs folder is the vagrants folder. And inside of there, you will find the various files for this example, notably the Vagrantfile. We'll open this up with Visual Studio Code and just take a quick peek at the labs folder in that Vagrantfile. You do not need to be an expert at Vagrant. In fact if you're not, I have this README set up with instructions for prereqs, how to install Vagrant, I'd suggest VirtualBox. If you have familiarity with another hypervisor, that's fine. You can plug it in with the provider option or use the environment variable VAGRANT\_PROVIDER. Down below then I have various listings of commands that I think might be beneficial, and it's rather long, but that's okay, I'm going to step through these. I just wanted you to have these for reference for copy/paste and also to look things up. If I don't cover something, I tried to hit on it here for you the common things you might want to do. Then what you have with Vagrant is a Vagrantfile that defines your virtual machine, kind of like a docker ‑ compose file defines an application and all the services involved, the volumes and networks, well this defines the virtual machines that you would like to spin up . So back to our dichotomy. We are focused on the node layer right now. All right, so inside of this Vagrantfile, I'm going to jump you right into the thick of it because I am using loops here inside of what is a Ruby syntax for configuration, a DSL built inside of Ruby. I have two loops that iterate over three and four numbers, respectively, to define up to three manager nodes and four worker nodes. Now that doesn't mean that you have to spin all of these up at once, but they're there if you want and you can change the ranges if you want to expand or reduce those. Inside of here, we are defining the virtual machines. I'm naming them m and then whatever number in the loop, so m1, m2, m3, and then for each of those I'm taking that node that represents the VM. In terms of the configuration for Vagrant, I'm specifying for the network and the hostname the values I'd like. I'd like to host name to be m and then 1, m2,etc., for ease of referencing. And then I also want to set up a private network and explicitly set IP addresses just to make life a little bit easier when it comes to joining nodes together. So I'm using the 99.20 and then 1, 2, 3 for managers, and 21 and then 1, 2, 3, 4 for workers. Workers are all named w and 1, w and 2,etc., and the hostname likewise. Down below, one thing you might want to change is your settings for your VirtualBox memory and CPU allocation. Depending on your machine and what you have available for resources, you might not want to allocate this much to each machine. You also don't have to spin all of these up at once, but if you'd like to tweak these settings you can. And then for the rest of this, I'm using the Docker provisioner. This, behind the scenes, uses get.docker.com if you've ever used that convenient script. So Vagrant is using this, as well as a few other tweaks to install Vagrant onto the virtual machine for you using what it refers to as a Docker provisioner. And I have passed no argument, so all I'm doing is provisioning to make sure that Docker is installed. You could go about installing this another way if you'd like. For example, I follow this up with a provisioner for a shell script. Provisioners run after the VM is created for the first time. In this case, I'm installing some additional software, tools that I like for debugging purposes. This list might expand. You can add your own as well. Just things I like inside the VM when I'm getting in there and poke around to see what's going on. And I can't recommend that convenient script enough. I think it's a great way when it comes to learning just to quickly spin up environments. By the way, I have a course out on Managing Docker on Linux Servers. You can visit that if you'd like to know anything about the installation process. I even cover the convenient scripts. They are only recommended for testing purposes, by the way, or for learning, not production environments. And then down below I have some additional notes. For example, links to the documentation for Vagrant, links to a tool to search for boxes. Boxes are what I skipped over at the top here. Boxes define an image, if you will, for the VM, kind of like an image for a container. In this case, I'm using Ubuntu 21 ‑ 04. Here are some other builds from Vagrant Cloud that you could choose from, or you could even search for another distribution or operating system, perhaps CentOS instead of Ubuntu.

Vagrant Up m1 w1 - Auto-provision Two Ubuntu VMs with Docker Engine Running

And the really neat thing is all I have to do is hop over to the command line, right where I checked out these files. I've got that Vagrantfile here, make sure it's in the same directory you're inside of, and just run the vagrant up command and specify what you would like to up. Maybe I'll bring up one worker and one manager node. Okay, I'll fire that off, and in a matter of maybe seconds or minutes the box will be pulled down if it isn't already, and then Vagrant is going through the process of provisioning the virtual machines for you to be able to access them over SSH really easily. Okay, here we go. You can see some of the packages are installing. Over on the left ‑ hand side Vagrant spits out the box that it's working on, so m1, w1, you can see the progress of each of them, and if you're curious you can open the VirtualBox GUI and you'll see information reflected here for the boxes that Vagrant has created, so m1, w1, prefixed by the project name or the directory name of the Vagrantfile, vagrants here. We can see we're installing Docker on the machine. All right, so the commands have completed there. If I want to check the status I can do that. You can see m1 is running and w1 is running.

Use the --advertise-addr Flag with Docker Swarm Init on Machines with Multiple NICs

We had our macOS node that we turned into a single node swarm, and now we've gone about the process of spinning up two nodes with Vagrant to make these into a swarm. It'll be a two ‑ step process of initializing one of them as a swarm node and then joining the other one to it. All right, we have a couple of ways to come about this problem. First up, we can just ssh into these machines. So we'll go into m1 first. M1 is short for manager1. We'll talk about node roles in a moment. Let's close this out here. And let's go ahead now and take care of creating our swarm. So first up, I want to look at what our status is, so docker info will tell me that. If I scroll up here, you can see, okay good, we are inactive at this point in time. That's great. Clear that out. Do you remember what command I used to create the swarm? So we need to type docker and then swarm. If you ever need help, just remember you can just hit Return without any arguments and it'll show you the various different subcommands, and we want the init subcommand. All right, let's fire this off, and oh ‑ no, we've got a problem here. So, this comes up when we have multiple IP addresses assigned to a device. In this case it actually shows us the two IP addresses that are causing problems, or we could have found that by running a command to look at the various interfaces on our device. You can see 99.201 and 10.0.2.15. So we simply need to inform the daemon which to use to advertise as the port for communication between nodes in the cluster. There we have two choices. And if you're curious why, I've gone ahead and opened up my hypervisor VirtualBox here, I've got m1 selected. I'm going to come to Settings, go to the Network tab, it gives me a bit more detail than I might get on that front summary screen that also has some of this information, but you can see we have one adapter set up as a NAT adapter. That's the interface that Vagrant is connecting through. And then we've got Adapter 2, our Host ‑ only Adapter, so that's our private network in VirtualBox. That is what we configured with the 99,201 address. If you'd like to confirm which is which, come under Advanced here and there's a MAC address available, 0800 is one of them, and then the other one, 0263 for 10.0.2.15. So that doesn't match. If I come over to Adapter 1, you can see this is expanded out now, 026384, that matches the MAC address for the 10.0.2.15, which is the NAT'd interface which we do not want to use. We want to use the one that is a private IP address, which would be Adapter 2. So that's how you can go about confirming which to use based on your networking strategy and setup for your particular VM in this case. Or if it's a physical machine, if you have multiple NICs it would be the interface you want to use to talk to your other physical machines because at the end of the day, our nodes have to be able to talk to each other on a common network. So, now that we know which one we want to use, we can simply then copy the IP address. And I can just come up here and run my command again. I want to specify advertised the address and just paste that address in here, the 99.201. And there you go, we've now got our cluster initialized. Quick quiz. What can I do here to check the status of this particular node in terms of whether or not that swarm mode toggle is turned on? Well, that's a simple call to docker info, and I should get an expanded output, and I do indeed have expanded output under the Swarm section. The swarm is now active. Notice that it is also a manager. The first node to join is going to be a manager, otherwise your cluster can't operate. And then there are other various settings that are defaulted here. And if I split the screen here, I'll ssh back in a second connection to my VM. I'll run docker swarm init again. This time I'm just going to grab the help for this. If you compare and contrast the init help and the various different settings that are up here, you'll start to see some overlap. For example, there should be a setting in here for task retention size, task ‑ history ‑ limit, there we go. It defaults to 5, and there you go, you can see it's defaulting to 5 right here. I also have the Default Address Pool. Come up here, default ‑ addr ‑ pool, that's how we can control that from the command line, as well as the SubnetSize here. We also have the Data Path Port. That matches up. So a lot of the flags that we could pass to docker swarm init will dictate the output of our info command here and really dictate the cluster level parameters that we're working with. These parameters are not all set in stone. For example, you can use the docker swarm update command to update some of them. For example, here is that task ‑ history ‑ limit, or retention. And then others are updated by other aspects of the cluster, for example networking. And many of these are advanced configurations that you'll be concerned with when it comes time to deploy a production instance. For now, getting started, don't worry too much about these, just know that they're there when the time comes.

Faster Node Management with Direct SSH Access - Leveraging Vagrant ssh-config

Okay, so I have 2 connections opened to my VM m1, and I hinted that this is one way to go to connect to the VM and control the initialization process of our swarm to interact with the Docker daemon. And while this works, there's a little bit of overhead involved in switching between VMs, or perhaps opening a second window like we had to. So what I want to do is back out of here, come back onto the host, and I'm going to close one of these windows. If you look at the files here, there's a host folder, and you don't have to run these scripts to follow along. I set these up mostly for my purposes. But these scripts do a few things. First up of interest is the host ‑ verify. All this does is just checks the configuration of what I'm about to set. There is no ssh config setup. And in terms of docker context, which is what we're going to use here to connect to our remote VM, you can see we only have the default. So that's a before snapshot. Now, what I'd like to do is just run the setup script. This will find running VMs, in our case, that should be m1 and w1. It will go ahead and dump out SSH configuration for those from vagrant using the vagrant ssh ‑ config command. And that's necessary for two reasons. It's faster just to use ssh itself. Vagrant has some overhead to connect via ssh. It's convenient, but it has overhead, so that's why this command exists. It's also necessary when it comes time to use the docker context feature, specifically, an SSH context. We want to have our configuration for that ssh connection stuffed away into SSH configuration files so that we can simply point at the right host and be done with it in terms of configuring a context. And I'll show you what that looks like because that's what we did last here. We went about creating two separate contexts, one for m1 and one for w1. And if we have more running VMs, this script would create more. If I run my verify script again, so first up, you can see the file does exist that I wanted to create here. This is nuanced to my SSH configuration. I point at this config.d folder and include all the configuration in there. So what I've done is just dumped out the output from vagrant ssh config into this folder so that at the end of the day, I can type the ssh command in m1, and just like with vagrant ssh and m1. Now it's a little bit slower with vagrant.

Transparent Node Management with SSH Based Docker Contexts

So that's what that first part does. The second part here for listing out docker context, I create two contexts, m1 and w1, and I use ssh for that context, and just point at the host m1, and I can do that thanks to how Vagrant writes out the ssh configuration. Let me dump out that file for you. So here is the file that we created. All we're doing here is giving the host a name, providing the host IP port, in this case using that other interface that Vagrant had set up. That's why I used Vagrant to generate this because these port numbers could change as we add more machines, for example, they're not going to be the same. It also specifies the correct user, and then most importantly, the identity file that Vagrant creates that's unique to this particular VM, it creates a secure connection. You'll notice that's in the same directory we're working inside of. There's a .vagrant folder. So right here's the repo we checked out, and were in the labs/vagrants folder. And inside of there, there's a .vagrant folder, and inside of there, nested way down deep is a private key for each of the machines. So there's w1's and there's m1's. So we just wire all this config into ssh, so it's ready to go. So when we set up our docker context, we just have to pass the URL with the hostname in it. Everything else is pulled from that config file. And so then the neat thing is you can see the asterisk next to default is the current context, we can change that. There's a use command, and we can just say use m1 or use w1. And of course if I list out the context, what do you think will happen now? Well nothing is going to change, and that's because I'm still on my host in terms of the client, and so the client configuration is what we're dealing with with context, and so that's not going to change. Now, the next part, let's say I wanted to get some information about that Swarm toggle to see if it's activated, what can I type in here?

Docker Node ls + Seamless Docker Context Switching

Alright. So that's the docker info command. And then from a remote connection over SSH, I can run a command as if I were SSHd into that particular VM, except I don't have to do that. I'm sitting right here on my host. If I need a second window, I can pop open a second window, can list out my context. You can see I'm still using m1 as the current context, that's pretty cool, and I could perform some other command like docker node ls, another way to look at the nodes in a cluster. And so, I hope you can see this is a mechanism whereby we can remotely interact with other nodes. And in the context of swarms and clusters, we want to do that for purposes of managing the cluster, managing the nodes, that lifecycle of the node itself, not necessarily for deploying any sort of software, that's the role of swarm itself. So just to prove I am connected to a different node, I will use the default connection again, and I will run docker node ls, this is going to be on my Mac, which we set up the other day with a swarm as well. And you can see we have a different host name here, a different ID as well, and these happen to both be leaders, which wouldn't happen in the same swarm cluster so this is all proof that we are just flipping between two different connections here. In fact, I could go ahead and use w1 instead, and if I type in docker node ls, what do you think will show here? So we are on w1 right now, it is not a part of a swarm, and so we're getting an error message back that says, hey, this command is a swarm management command, this node is not part of a swarm, therefore , there is no utility in running this command. It does say we can run a docker swarm init to create a cluster or we could run a docker swarm join if we wanted to connect to an existing one, so let's do that in the next video.

Visualizing Docker Contexts to Simplify Multi-node Management

So this is our current topology. And to take the next step to join w1 to m1, I'm going to leverage context to show you how it can be really simple to do without needing to SSH into the VM. I think contexts are a very important part of managing a cluster. When you need to get in and out of nodes really quickly, it's probably the best way to do it. So keep in mind all of this right now is on one machine, my Mac with Docker Desktop installed, that's got a single node swarm, and then we have our two vagrant VMs, one of which is initialized as a single node swarm as well. And now on the left ‑ hand side here, I'm going to pull up what looks like a prompt and start out with a little bit of the code that we were working with, color ‑ coded to help understand context. These are going to be vital going forward because I think they're a great tool to manage the nodes in a cluster. Even after you have the nodes joined together, this is still valuable to get in and out really quickly without needing to fully SSH into a machine. It's almost like a transparent SSH connection, actually. Okay, so up at the top here, we have our vagrant ssh config. Remember that just writes out the configuration to access the vagrant VMs, puts it in a convenient spot, and gives each of our VMs a friendly name of m1 and w1. That way we don't have to refer to them by IP address, we don't have to provide the username, we don't have to find a way to provide the private key file to access them, et cetera. It's all wrapped up in that config file for us. And then the next two lines are what it would look like if you ran the context create command instead of using the scripts that I had, and if you have any trouble with the scripts, you can use this sample code right here. It'll be checked into the course repo; another reason why I wanted to show this here. So we'll create two contexts, a blue one, and a green one, and a cyan one, and then I turn around and list out the Docker context. First up is our red or pinkish context. That points out the Docker Desktop single node swarm right now. That's just the default that's hard coded. You can't change that. And then next up we have m1. Essentially we are aliasing the SSH protocol pointed at that friendly name of m1, that ssh://m1, we are aliasing that with m1 for a context in Docker, and then we do the exact same thing for w1, the cyan ‑ looking context . Okay, so then we used these contexts, and I've captured the essence of what we did with them here with some modified commands, but they'll make sense in the context that we just ran. So the very first context, the default context, is the one that's set by default until we change something. So I'm using the docker context show command, and you can see it is default. Anytime you run that show command, you'll see whatever it is that you have selected for the current context. Because default is our current context, you can see them when I run docker info down below and grab some of the fields out of the result. We are pointed at our single node swarm, so we have a swarm that's active. You can see the IP address. We can see there's only one node in the cluster. And we can see the name is docker ‑ desktop confirmation that our context has pointed out the location we think it is. Now if I want to connect to m1 instead, what do I do differently as the very next command? Well, let's pull up the next part of the script here. So, we just need to run a use m1, and that we'll switch us over to the m1 context and point us at that m1 node. Running the exact same command as above, you can see we have an active swarm as well because that's our single node swarm. There's the IP address, and that should look familiar as the very first manager node's IP address if you look in the vagrant file, and then you can confirm the name of the machine is m1. And I have one last command here. This would be a little bit different than what we did up above. Let me show you this here. So I'm specifying the context with a command ‑ line argument here to the Docker CLI. So docker, and then a ‑ c w1 selects the w1 context, which will point us at the w1 node. The rest of this command is the same as before, so you can see we have an inactive swarm, and the name of the node is w1, confirm that we did switch contexts. Now, I've got a question for you. If I were to run another Docker command after this and didn't specify the context at the command line anymore, which context would I interact with next? So imagine dropping off that ‑ c argument to the last command and running it again. So when I specify the context at the command line as an argument to the Docker CLI, I haven't switched contexts then, and so subsequent commands would actually fire off against m1. That's the last context that we switched to with the use subcommand. The neat thing about this is if you have two or three servers that you need to fire off commands to really quickly, you could just loop over them and spit out the command, add one argument to it, and you can execute your Docker commands. Don't worry about what context you end up using at the end, it's going to be whatever you started with because you didn't switch any context along the way. Perhaps you can recall the docker\_host environment variable and its affiliated family of other variables to configure how you access a node. And perhaps you could even remember before we had the ability to use SSH as a protocol to access the node. You're using TCP or perhaps trying to run a secure TCP socket. If you did that, you know there are quite a few arguments that have to be passed. They're still there, they're just all hidden behind the context for you. That said, I can't recommend enough using SSH contexts whenever possible. And I say that because A, it's secure, and B, it's a lot easier to configure, as you can see the commands up above take care of that for us. Just have to configure SSH, and if we have that already, well, then we don't have to do anything else than use SSH.

Joining Nodes Concisely with One-off Commands via Docker Contexts

Let's apply what we learned about context to take our single node swarm and turn it into our two node swarm. All right, so I'm sitting back here on my host, and I want to show a little change I made to my environment. First off, I have truncated my prompt to only show the current folder I'm inside of. So I'm still sitting next to our vagrant file just to give us more space because I wanted to add a feature, and I created a toggle for this so that it enables this feature, that will show me the Docker context that I have selected , in this case, m1. So if I use a different one, you can see it switches to default. And, of course, I can turn that feature on and off as I see fit. So if you see that throughout this course, you'll know what that is. Primarily, it's a little hook inside of my shell that reads the docker context inspect command, not passing any args, which will just give me details about the current selected context. In this case, I'm extracting name. And then if there are any overrides for the host, for example, I'm grabbing that information as well, for example, with docker\_host. Yes, I'm grabbing information from there, that's where that comes from. I'm doing that so that you can see as we switch between machines which machine we execute commands upon because otherwise it can be kind of confusing from the host to know what exactly you're targeting, especially if you start scrolling back in your command history and looking at what you executed. I say all that because this is actually going to be useful to us. What I would like to do is not do the same thing as m1 in terms of setting up w1, so note vagrant ssh w1. Instead, I would like to use what we just talked about with context to quickly toggle on swarm mode by joining it to the m1 single node swarm cluster. What's the first thing I need to do here? Because I'm making changes to the w1 node, I will switch to that context. Chances are, I've got more than one command to run against that, so it would be helpful to have that as my current context. First up, I'm going to need the join token. So even though I just switched to w1, I actually need some information from m1. Now, I'm not going to do that by switching because it's just one piece of information that I need. Instead, I'm going to arrow back here, and I'm going to put in the global context argument to override and say m1 instead. And then I'm going to say I would like the join ‑ token. And in the case of w1, I want this to be a worker. So I'll fire this off. This will connect out to m1 and ask for the join token to become a worker, so that's a role. Nodes that are assigned to the worker role, they're essentially limited to just performing tasks. And managers, on the other hand, get to decide who performs the tasks, and by default, managers also will perform tasks as well. consequently, because of the power of a manager node being able to make decisions, the token is different because you wouldn't want anybody to have access to that token, whereas the trust necessary to add a worker is not quite as big of a concern as adding a manager. Now, the nice thing is this is all formatted for me. And then I can just run this command here. And thankfully, we're pointed at w1 for a context, so you can see the node is now joined as a worker. To confirm that, I can grab the system information, and scroll up here, and you can see we now have an active swarm node. I might also want to look at the nodes that are in the cluster now, and it seems reasonable to assume I could just run docker node ls, but I can't, and that's because I am not a manager. In my case, because I have access to a manager node, I can just connect over and run a one ‑ off command on m1 to take a look at the nodes. And there you can see we have the HOSTNAME column with m1 and w1, where m1 is a leader manager and w1 is not a manager at all. Now to really confirm this, in the next video, I've got a little piece of software we can run that will give us a visual representation of our cluster. Let's take a look at running that next.

Visualizing Cluster State with the Docker Swarm Visualizer (dockersamples/visualizer) Running in a Standalone Container

Most of the time, the command line is just fine for learning. You can even list out nodes here and get somewhat of a visual representation of what's going on with our cluster. Though, you do have to drill into different aspects to collect up all the pieces of information. And you could do that with a series of commands chained together, but I prefer another tool referred to as the Swarm Visualizer. This used to be called \_\_\_\_\_ I think swarm ‑ visualizer. It's now in the dockersamples repository under visualizer. What this is going to give you, if you take a look at the Git repo for this, is a nice visualization of the machines in our cluster, as well as, eventually, what is running on top of them when we get to the next module and talk about services. To help you get started with this, if you come over to the exercise files, I'm sitting inside the viz folder. Print out the repo directory this is inside of. And inside of here, you can see there are a couple of different files. Focus on the viz ‑ container.sh. It's a script that has a simple call inside of it to start up a container with Docker. And, specifically, this needs to be run on the manager node on your cluster, depending on what your setup looks like. Because just like the docker node ls command does not work on a worker node, because it's not privy to that information, you need access to the APIs on the manager to be able to get the data to show that visualization. If you notice, my context is set to m1, which means I'm ready to go. So I'm going to clear out the screen here. And one of two things, I can execute that file or I copied and pasted the command here. What we're doing here is creating a container the traditional way. So you should be familiar with this, docker container run, or perhaps you are used to docker run. We are specifying the m1 context to be safe, though I am already using the m1 context, as you can see here. This will be a detached container, so I want it to just run in the background because it's going to be a visualization. And then I've got the port that we access it on. I'm just mapping it right over to 8080 on the host. And then it needs access to the Docker socket to be able to get the information to display the state of the cluster. And of course, the last line is the image itself. So, I can run that. Take a moment to pull down the image. And it looks like it already has the image. So now I can come over to a browser, load up port 8080 on the manager node using the IP address right now, and you can see we have a nice visualization. Nothing running inside of it, which might be a little bit weird because we do have a container running, but it's not going to show up here because it's not a part of the Swarm ‑ managed services. It's actually running with the standalone capabilities of the Docker Engine. So, another thing I like to point out when doing this is there's a big benefit to the swarm mode toggle, and that's that you can continue to use Docker the way you've always used it, as well as this new way with clustering machines cùng. If we come over to the command line here though, print out my containers, you can see pointed at the m1 context that we have our visualizer running. So we'll leave that there, and now let's make some more changes to our cluster and take a look at what happens with that visualizer.

Joining Worker 2 with Real Time Web GUI and Terminal Monitoring

Alright. We've got our nice visualizer up on the right. I've split the screen in a third and two thirds here so the terminal has plenty of space, and on top of the terminal window, I've got a watch running right now that just prints out the list of nodes with docker node ls . I have an alias for this. If you want to see what this looks like, I can take a look at the function here for you. The most important part here is that I am running a docker node ls and passing the context along with it to make sure that I'm connecting to the correct context. If you'd like, the watch command is what I'm using, I installed it with Homebrew. No ‑ title drops the date and time and the header that explains the command that's running, which is unfortunate, but it saves some space. It also got here, ‑ n 0.5 means refresh every half a second, and ‑ d means highlight differences. So if a value changes, for example, ready on one of the nodes, for example, it goes offline, when the text changes to something like perhaps down, you'll see a flicker in the contrast of the pixels around that area to draw your attention to it. Alright, so let's fire this back up. That will be running on top, and then down on the bottom is where I'll execute commands and we will have then two ways to look at this and I want to give you a taste for what you might think of the graphical representation or the terminal representation or textual representation. I like both of these actually, but let's use them to see what happens as we add another node to our cluster. Do you remember what I can do here to fire up another node? Well, we still have some available so why don't we fire up w2, give that a second and I'll come back when it's done. Okay, w2 is up and running. I can take a look at the status that I have configured if I'd like. Great. And if I want to be diligent, I can go ahead and run my host setup script that will recreate my SSH configurations and add a Docker context for w2, but you can see up on the top there what happens when I am context. You can see a flash there with the contrast, that's what I mean by ‑ d. Okay. And if I want, I could even take a look at the output of the new context, and look at that, we've got our w2 context. Alright. Now, what do I do next if I would like to add w2 into our cluster? Well, first up, I need to make sure that I am in the context of one of the manager nodes. So you can see here at the terminal prompt I have m1 so I'm good to go there. After checking that, I know I can run commands to modify the membership in the cluster, and if I don't remember what exactly the join ‑ token is, I didn't write it down, for example, I can just run the docker swarm join ‑ token command and then I need to provide worker or manager. And so in this case, we're talking about a worker and that will spit that out. I believe there is a quiet mode too, yeah, there you go, quiet mode where you can just spit out the token itself if that's all you want, but I like the more verbose mode because I can just calculate this the whole command, it's ready to go, it's even got the address to connect on. By the way, this is why that advertise address is so important when you're initializing a cluster, and also, when you're joining other managers because managers are the nodes that are connected to by the workers, so this particular advertised address has to be available to them, and thankfully it is, in this case. Alright, copy that, and then I need to switch context here, and we'll switch over to w2 and paste this in. Now before I run that, I want to remind you to pay attention to the right side and above and see what happens. Look at that. Really quickly w2 node is joined to the cluster and that's, of course, because it's booted up already, but do you see how fast that was? That was impressive.

Simulating a Power Loss and Observing Node Status as Nodes Come Back Online

Now let's take a look at when something goes the opposite direction. Let's take a look at when something goes down. You might have noticed this happen if, for example, you rebooted your machine while your VMs were running, and they are no longer running, I'll simulate that with a vagrant halt, which is the equivalent of a stop command or a shut down . In this case, I will specify w1 as the node. Now I'm going to highlight the parts of the screen to watch here as that VM shuts down. There it is. Did you see that go down? Let's do worker 2 here and see what happens. And there you go, did you see that, we've got w2 as red, and then up in the top, did you notice anything happen? Somewhat subtle even with the difference flashing, but you can see in the STATUS column for w1 and w2, they're both Down, whereas they were Ready before like m1. And of course we could simulate these coming back online by just bringing them back up, w1, w2, give that a second here. Pay attention to the impact on our two views. Hey, w1 is ready, and w1's ready over on the right ‑ hand side too. The red dot is still there for w2 though, almost there. All right, did you see the difference flash up on top and over on the right ‑ hand side, everything is green again. So, this is one of the reasons I really love Vagrant, the ability with which you can play with the VMs, turn them on, turn them off, simulate all sorts of conditions that you would have a difficult time simulating in reality, short of perhaps pulling a plug out of a wall to simulate a power loss. Don't do that though. The nice thing is you have virtual machines, and that gives you a nice way to see what happens with your cluster. You can try out a lot of different scenarios.

Joining a Second Manager Node

I'm showing one more node here. This time I want to join another manager node. So I'm bringing m2 up. I'll pause this and come back. All right, that's done. And then over in a new tab here, I've already run the join ‑‑ token command to get the manager token because that's different from the worker token. Copy that, and then I'm going to hop back over to our first window here and down below. Right now I'm on the wrong context, so I'm going to need to switch context to m2. Not found. That's because I haven't created it yet. Let's create that real quick. All right, up on top, we had the big change over, and it looks like everything's running now. Now we should be able to switch over to our m2 context. Wonderful! And then I should be able to paste in the join command with the token for a manager token. Then pay attention above and to the right when I do this. So m2 took a little bit longer to come online, and, sometimes I notice this, the Visualizer UI will have something that's not quite right, like, for example, the RAM amount here. Just refresh if that's the case, and it should pick up any changes that it missed. Then over on the left ‑ hand side, you can see everything is ready to rock and roll. Now in terms of the manager nodes, we have m1 is the leader, and then m2 is reachable. So of the managers, 1, the leader, is the one that's really in charge doing the heavy lifting. The others are there to step in if something goes wrong and become the leader. And to do that, they have to spend their time maintaining the state of the cluster precisely so that they can actually step up should the time come. Also, by the nature of having multiple managers and the ability to fail over, you introduce the capability to fail over, thus reducing downtime as long as you have more than one manager, and a minimum of three is recommended.

play-with-docker.com - How to Setup a Learning Lab with Free, Multi Node Clusters in the Clouds!

Briefly, I would like to give you another possibility if you would like to have a multi ‑ node setup, you only have Docker Desktop or some single node available, and you would not like to have to deal with virtual machines. Well, you're in luck. There's a tool called Play with Docker released years ago, and it's been around for quite a while and it's been really cool. I remember running an entire workshop off of this for participants, and it was really a great way for people to get up and running with a cluster. So, just come out to play ‑ with ‑ docker.com, with dashes in between the words, click the Start button, and a session will be started up. These sessions last 4 hours. So, one of the limitations of this is that it's better for short ‑ term little learning exercises that you have, not so good for long term. And of course, that's to try to curb some of the abuses that would come if they had a system online that you could host applications from. Because you can actually access these apps on the web, people would start redirecting requests to them from their websites. So you probably want to rely on scripts or copy/paste code snippets that you can use to create your environment. Or, there's a little wrench here that you can click, and it pops up templates that you can generate. Because what this does is, it can more or less create instances on the fly. They're simulated instances of machines. So I can click this button over and over and over, and I get three nodes here. And now I could go about the process of initializing them with docker swarm init and join. But if I don't want to do that, I'll get rid of these nodes here, instead, I could come over here and say, hey, I'd like this 3 Managers and 2 Workers template. Generally that for me. Get everything going for me. You can see all the nodes are named nicely. We've got manager1, 2, and 3. We've got worker1 and worker2. We can just hop right in, take a look at docker node ls. And you can see all of our nodes are connected together, so the swarm initialization and joining the nodes is all done for you. So, a good way to just get up and running with a very basic exercise and actually even faster to generate a swarm for yourself. Though, you could automate this process with something like Vagrant. You could have a script where the nodes are joined together. That wouldn't be that difficult to write, and it might even be a fun little exercise if you'd like a challenge.

Promoting, Demoting, and Inspecting Nodes

Why don't we take this for a test drive and give something a shot that we haven't done yet. Let's look at node, promoting and demoting. Workers essentially run our applications, managers coordinate who is running our applications, and by default, managers can also spend their resources running applications. Though, as you build a bigger cluster, you'll want to take that capability away from those managers. That said, maybe at some point you need to promote a worker to a manager, for example, may be one of your managers is lost for some reason. To maintain a quorum, you might need to bring a new manager into the picture. Let's just look at what that process looks like. So I've got a Docker node ls here. If we take a look at the docker node command, there are other sub commands available. Now we've already looked at the ls sub command. We also have a promote sub command. So why don't we take worker1 and promote worker1 into a manager node. So just up arrow here, and I'll type out promote and tab completion is a wonderful thing when it comes to the Docker CLI. If you don't have that installed, make sure that you get that set up for your particular environment, it makes life a lot easier with all these sub sub sub sub commands. So I can run that, get some help for it, and you can see we just need to specify which node we want to promote, so it's a one ‑ way direction to promote. We don't need to specify any arguments here. Let's just take worker2 and promote to a manager. And that's how easy it is to promote to a manager. And you can see over on the left ‑ hand side, it looks like managers are marked with a little person icon, almost like perhaps a supervisor watching over you. And if I do a docker node ls, I can confirm that that actually happened, the promotion, can see worker2, which is not exactly the best name anymore, but it suffices, is reachable as a backup manager now. If I want to get some more details, by the way, Ctrl+L will clear this out here and get Docker node and then inspect as a sub command, and then to that, I can provide worker2, a lot more information about this worker in here, and I should be able to scroll up and see somewhere in here, here you go, I have the role specified as manager and its availability is active. And of course, if I wanted to do the reverse, perhaps I was able to fix the manager that was offline, and I want to convert worker2 back to just a worker and it's just docker node and then we've got the demote command here . And again, there is no parameters to take other than the name we would like to demote, and there you go, it's been demoted. Let's take a look at our node ls and you can see it no longer has a manager status, and therefore, it's just a worker node. And if you want to confirm that with an inspect, we can just up arrow here, scroll way up to the top here, and again, we can take a look at the spec and we can see that the role here is worker.

Connecting to play-with-docker.com over SSH - And, Even Better, a SSH Based Docker Context!

One more thing before I move on, there's a little button here where you can click to get access to this environment, if you want to ssh into it, so you can copy that, you can come right over to my command line, and paste that in here. I'm prompted to accept the fingerprint, and I am now connected without even a password, but if you think about it, this big long blurb that you have to add to connect to this node is sufficient in terms of a password for the purpose of learning. So, in some regards, you can actually have access to your local file system and work with a node remotely over ssh in this case. In fact, if you really want a challenge, I haven't tried this yet, but we could try to create a docker context for this, set the host to that copied ssh command, come back over here, and we'll change this to indicate the protocol is ssh, so ://, give this a name of pwd for Play with Docker. So let's try to connect to that. So we use pwd and node ls. Hey, look at that, that works too. So now I'm on my actual machine, I have all my commands that I'm familiar with available to me. I don't even have to ssh into that node. But this is a really cool feature, it does give you the ability to work with files on your local file system then, so you wouldn't have to copy and paste things into that other environment. But keep in mind this session is going to expire. In fact, in the time I've been showing you this, I've already lost 14 minutes of my 4 hours. So just have stuff saved in a file and ready to go if you'd like to try something out new again. The nice thing though is once you're done, you can just hit CLOSE SESSION, and everything is nuked. GOT IT. It brings you back to the start page, you can hit Start again at any point in time you want, and then you've got a brand new session here, you can add your new instance, this way, if you wanted to try out maybe join nodes together manually. While I'm at it, I probably should get rid of that context consider that that session was terminated.

Deploying Services to the Cluster

Abstracting the Challenges of Multi-node Container Deployments with Services

Now that we've spent some time talking about the nodes, and I said there's a nice division between nodes and then the other big aspect of what we run on a cluster. We've got several different clusters setups designed here with a variety of technologies. You can take your pick what you like because now we're going to focus on what it is we run on top of those nodes. And the first thing we want to take a look at is how we can orchestrate containers. Just like on a standalone node, we want to spin up applications that are running in little isolated bubbles. So, for example, if two applications have the same library dependency, they don't conflict with each other. Fortunately, in this case, with a swarm, there is no conflict in terms of that functionality containers of a container because each of those has an image it's built on top of that's isolated regardless if it's on the same node or another node. But then there are other challenges when it comes to what we want to run on our cluster. For example, I mentioned perhaps two containers on the same host need to access a port, and perhaps both of them are web services, and maybe they want to both go for the web port. Well, we know that containers solve that problem somewhat by giving each of these apps their own virtual interface to the network, so combine to port 80 each if they'd like. But then if these two applications need to talk to each other, somehow they have to be routed together, and we can do that if they're on a single node. But what happens if they need to talk to each other and they're on different nodes? Well, in that case, we have to create some sort of virtual network that spans across our nodes to give connectivity to our applications. Of course, a more basic problem if I want to run another container, where is it going to be placed? Should it go here? Should it go here? And it needs to grow in terms of the resources that it uses, is there space for that here? Maybe it needs to go here instead. Or what about here where it kicks out some of the other containers? And how do we work with the tooling so that it's got the same workflow as standalone where we just keep adding containers and we don't really care about this placement concern? Of course, we could run out of resources on a single node. That's why we're talking about multiple nodes. But we want the solution to be similar to what it was with a standalone instance of Docker. Another factor, what exactly are we running here? Perhaps a lone web app. And how exactly does that get distributed? Perhaps two separate instances. Does it need some sort of database? And how does that get distributed so that we have failover? Perhaps a standby in a primary. Perhaps reasonable risk requires some sort of model that's kept in memory, and if it's complex enough, it might require some sort of API sitting in front of it that gets distributed across the cluster as well to handle the load that the loan web application talks to that then talks to the risk model. Because what's going to happen is we have somebody come along that says, hey, I need a car. So they come in for a loan, and they put in a request for an application. The web app might need its database to read some information, so it can maybe read from the standby that's right next to it. And then maybe you need some information about the risk involved, and so it needs to find where is that risk API at. Oh, there's an instance I can use. Okay. And then that risk API might need to say, hey, where are my models at? All of these questions asked are a form of service discovery. Even if it's a person outside of the organization, it's an external query for where they can go to fill out a loan application. Or it might be an internal query about how to find another service that's running inside of a cluster. These are just some of the challenges that Docker had to solve and make transparent for us, and the solutions to the container side of the equation is wrapped up in what is called a service. So let's take a look at what it's like to deploy a service and compare it to what it's like to deploy a container.

Full Auto Swarm with Standalone Container Visualizer to Convert into a Service

All right, I am sitting inside of an autos folder inside of the labs folder. See the path right there, and you can see the files inside of here. There's a Vagrantfile in here. And I wanted to show you this because in the process of understanding a service, I set up another Vagrant environment that's fully automated, includes adding that containerized version of the visualizer, and then I want to contrast that with the equivalent version as a service. So first thing I'm going to do, I want to look at the Vagrantfile briefly. So under autos and the repo, there's a Vagrantfile most notably. Really the big change here is that I added calls to provision, and then I added scripts for those underneath the scripts folder. One script will initialize the swarm with m1 when it comes online, and it'll write out two tokens, the worker token and the manager token out to disk. And then as the other nodes come online, worker ‑ join and manager ‑ join scripts are run to join those nodes to the cluster automatically, and they use the tokens that are written out. The nuke ‑ tokens is just reminder after the fact. You might want to remove those. It's not necessary because they'll be overwritten if you ever reinitialize the swarm. And then the most important of these is the one to create the visualizer. I should be able to come over to the command line here, and I should be a little vagrant up, and m1 needs to be created first before any others. So the tokens are written to disk so that the rest of them can join through the cluster. By the way, while this is starting up, in the browser I opened up a tab and pointed it at 101, which is the IP. I changed the IP addresses from 201 and 211 and 212 etc.. I changed them to the 100 range. I wanted to preload this tab so that the second the visualizer is available we can see it. Once Docker is done, it's going to move on to the provision, or to create that, and I want to be able to see that before the swarm is initialized. And that'll happen really quickly, so why don't we split the screen here. Hey, look at that, and m1 is initialized. That's how fast that happens. On to the rest of the nodes. And by the way, I actually started up with vagrant up alone, so all the nodes are starting right now. M2 is actually in process. To guarantee that m1 is done, I'm running sequentially so that m1 finishes, then m2, then m3, and so on. And let me speed that up here. Now let's talk about turning this standalone container into a service.

Symmetry of Docker Container Run --help and Docker Service Create --help

So we'd like to convert our docker container run statement into the equivalent with a service, a service being the representation of a unit of compute. Part of the reason for containers being called services now is because there can be more than one container per service. So a container is a lower ‑ level concept that is still present when it comes to dealing with Docker Swarm. A higher ‑ level concept is that of a service. One thing I like to do is just take a little peek around the command line. So first up we have docker container, and you've probably seen this one. It's a very long verbose list of a whole host of commands. Notably, run is how we can both create and start a container in one operation. If we look at that, you can see there are a lot of options to pass. I would recommend that you read through this at some point if you haven't for the docker container run command, but more important is to go over to the docker service side of things. This one is much more abbreviated in terms of operations. Although it has grown a little bit over time, I doubt will grow extraordinarily long like the other lists because most of what's necessary in terms of representing a service is already captured by this set of subcommands. Arguably, this might be better designed than the original docker container subcommands because those are from way, way back in the early days of Docker. Also, this is representing a slightly different concept in that we're really talking about a definition for one or more containers at the end of the day. Now if you had to guess which of these options do you think is roughly equivalent to what we saw with docker container run, it's the create command up at the top. Now it's a bit different. We have to create command with docker container, and it doesn't do the same thing, but create here creates our service, defines it, and then Swarm is allowed to take that service definition and make it come to fruition. So let's take a look at the help for this as well. And you can see there's a long list of options here too. And in fact, if you compare the two side by side, you're going to find a lot of similarities between this and what we had with docker container run. This is a diff here of both outputs. I've got a little script in the repository if you're curious about trying this out yourself. And as you can see, the yellow, which is the overlap, there's quite a bit of it, and then the red. Oftentimes you'll find that sometimes something has been renamed or the description text has been changed a little bit, and that's what some of the difference is in here. But as you can see, there's quite a bit of overlap. Again, this is a list I would recommend that you read through over time or some time sit down and take a look through. For example, there are ways to control the DNS setup inside of your container. You don't need to know the exact settings, but just know that you can tweak DNS. Then when you need to do so, you can come find the relevant options.

Porting and Running a Visualizer Service

All right, so in the interest of saving some time, I've got a pre ‑ canned port of the container run for our viz service into an actual service, and that's inside of this convert ‑ viz ‑ to ‑ service. It's right next to the m1 ‑ viz ‑ standalone ‑ container shell script. So, first up, we have our docker service create instead of our docker container run. We have a ‑‑ detach up here. We can add that down here if we'd like. I chose to leave it off. For publishing the port, I've just basically copied the exact same value down to the script below. To make sure that this doesn't cause any conflicts, why don't we switch this one over to 81, so this will be mapped to 8081. Since we still have that standalone one running, we can leave it running. And then instead of a ‑‑ volume flag, there's a ‑‑ mount flag. And this is a bit more verbose using a long format, or syntax, for specifying the options of what exactly the volume is mounting. So up above here, this is implicitly a bind mount that specifies a host path and then a target path inside the container. Well, down below, we have the same information. We say type=bind, and then we have our source on the host, and then our destination is inside of the container. The last piece of this then is just our image. And now while you're converting this, how would I know, for example, that mount is what I have to use or that I can go ahead and use the same value for publish? Well, in addition to the command line help, there is a set of wonderful documentation around docker service create and then the docker run command. Now docker container run is the same thing, but this is embedded inside the docker run and Reference. Make sure you look for Reference out on the Docker docs. When you find pivotal commands, they will have a whole bunch of documentation along with them. For example, creating a container and running it. Look at all the information. We can scroll through a lot of explanations of inside of here. For example, here's how we can deal with volumes, bind mounts, using the mount syntax instead actually with the container run command. And then on the flip side of things, over on service create, we could look for an equivalent section, and here we go. We've got Add bind mounts, volumes or memory filesystems. And under here is where you can see how volumes get transformed into mount. And if you scroll down here, you'll see some of the options I was just talking about. For example, the type of mount, bind mount in our case, the source, the destination, what these things mean. There's even a readonly argument that we haven't used. And then there should be something out here for ports. Yep, if we scroll down here, there should be a section for ports, and this will explain that we are sticking with the short form syntax. There's also a long form syntax for this, like the mount long form syntax, but in this case we can use either, and so I'll stick with what we had originally. So the next thing is with that help that we have the command line and online. We can go through argument by argument and decide how we want to port things over or how we have to port things over and come up with a new command to use. And then you can either copy the contents here of this file or you could execute the script. I'm going to split the screen here. On top, I've got my Standalone Visualizer so we can watch what's going on here. And then on the bottom, I'm going to go ahead and paste in the command that I copied. And let's fire that off and see what happens. So because we did not detach, we're going to get some information about the progress of deploying this service to our cluster. And if you look at that, our service just showed up on m1. First, let's just check out and see if it's working. I'll come up here on top, make a new tab. Do you know what IP address I need to go through here and what port I need to use? So this is m2, so it's going to be 102, and then it's 8081 is the port that we are using. And hey, look at that. We now have our visualizer that's running as a service instead of as a standalone container. In fact, we have both of them operating right now. See, can refresh both of them. We don't need both of them, but it looks like things are mostly operational. Now you might not have got as lucky as I did because there's one piece of information missing when I ported that script over, and I did it reluctantly. It's an implicit argument that is not so easy to understand. Join me the next video so I can talk about that.

Docker Service Update --force to Change Placement without Changing Our Service

As I mentioned, something is missing that could randomly cause problems for us. The first thing I want to do to elicit what this is is trigger the scenario that is the problem because right now with this visualizer service running on one of our manager nodes, it's not going to be a problem. I need to get this to operate on one of our worker nodes instead. Now, there are many ways I could go about doing that. One thing I'm going to do is show you how you can trigger your tasks and containers to be placed elsewhere, if you so desire, without needing to really change anything about the services deployed. Over at the command line, the first thing I want to do is run a docker service update. I'm not going to change anything though. Instead, I need to provide to this, well, I guess we could grab the help and look here. It's always a good way to go about figuring out what you can do with a given command. I need to provide the service, and I also want to provide one of these myriad arguments. This is the argument that I want to apply as well. So even if there aren't changes to the service, I want to force it to be updated as if there were. Imagine if I deployed a new version of the visualizer, for example. So I can run my docker service update, and then I need to provide the ‑‑ force flag, and then I'll specify viz. When this happens, what do you think is going to take place? Well, maybe already gave it away, but when I do this, this will trigger, hopefully, a replacement of that task and container on a different node. And I'm saying task and container, and I know I haven't explained those yet. A task is, if you will, a desired state for a container that's operating somewhere on your cluster. So what I'm going to do to see if this worked is switch away from the service ‑ based visualizer to the one we have just running standalone on our manager node and see what it shows in terms of where that service is at because this is the one that's not a service, so it can operate independent of how the cluster is actually joined together. Remember, this one's on port 8080 Okay, so you can see that actually redeployed to the same node, which means this should be operational again. Yep, okay. So let's try that again. Sometimes you have to do it a few times to get this to actually move around, if you will. So I'm going to come back over to the standalone one. Just watch what happens here. Man, it looks like I am not getting lucky here. Let's spin the wheel one more time. All right, third time is the charm. You can see we've moved over to worker 3.

CLI Task Monitoring with Watch Docker Service ps --format

Now I'll have to say this is quite ironic. I turned my computer off last night, and when I turned it back on, as the managers came online, then they went ahead and rescheduled the work onto themselves. So, once again, viz is sitting here running on a manager, which is not what I want. But that's good because I actually wanted to show you an alternative to the force update on a service in terms of getting your task to be placed somewhere else. And at the same time, we're going to abandon the UI. I want to show you a different way at the command line without this UI. And to do that, I want to show you a reason why I'm going to do what I'm going to do here. I'd like to SSH into the Vagrant VM and execute a command directly. And the reason for that is, I'll show you, we'll run the time on this here, docker service, and we will do a ps. Ps is listening out your tasks, or you can think of them as containers. Remember, they're a desired state for your containers, so they might not yet exist, and the task is the concept that represents what your container is going to become then. Anyway, if I run this locally, it takes about half a second to run. If I run this over and over and over again, it's half a second each time. And I'd like to run this at least at sub ‑ second intervals if possible. So what I want to show you is what it looks like to run this command right on the manager itself. So I am going to SSH into m1, and I'll run the same docker service ps and then viz as a service. But the important thing is if I run the time on the top, you can see this is a lot faster overall. We're looking at speeds that are 10 times as fast. So I can't actually run at sub ‑ second intervals here because what I want to do is run the watch command. And to this command, I'm going to pass a few arguments. I'm going to hide the title, ‑ d to highlight differences, and then we're going to do 0.5 on the interval. So that won't tax my CPU as bad. And then I'm just going to run the exact same command as before, and then we can watch here as changes happen. But one last thing I want to make a modification to on this command and help you understand a little bit about what's available here to you, I could zoom out here, and that probably would work, but that's getting really small for you to see, and I'd like you to be able to see what's going on here. So what I'm going to do is truncate out the columns that I don't want. Let's take a look at that next.

JSON and Custom Table Output with Docker Service ps --format

So I'll just Ctrl+C to quit out of the watch. So let's try this again here, docker service ps and viz, and I'm going to drop off the watch right now just to avoid any ambiguity. I'm going to add the ‑‑ format flag here, and I'm going to paste in a string. There we go. And I'll run this, and then I reluctantly left one thing off so I could talk about this. You can see we have a nice table here, but we don't have any headers on the columns. If you want to get those in there, and you want the table to be evenly formatted, no matter the spacing of the values of the columns, well, throw the word table on the front here, and you'll get back your column headers and nice consistent columns. So what I've done is I've effectively cut out two columns. I've cut out the image, which takes up a lot of space and isn't really important right now because we're not changing it. And I've cut off the ports. Everything else is the same. And, by the way, if you're curious about what exactly you can include in that format string, you can definitely refer to the docs. You can also pass a format string with a JSON and a dot inside of the template. And then I'm also piping to jq right at the end here to make this formatted nicely. I installed JQ on the VMs as part of my provisioning process. So when I run this, I'll get a linear list, if you will. I get an object ‑ like format, print JSON format to be specific, with each of the records. And then it's this casing right here, and it is this set of fields that I have to choose from up above when I'm producing any format string. One of the inconsistencies is that the ID, the D in it, is capitalized as well. So, again, all of the Docker commands have this available. It's going to be very helpful when you start working with a cluster and you start to get lots of different types of objects and you want to control, maybe limit, what you see about a given object and focus just on what you care about because there might be a time later in the course when we change the image, for example, in updating our application, and we'll want to bring that back then. Maybe we want to nuke something else at that point in time. Maybe we don't care about the ID at that point in time or maybe the name won't matter to us, like in this case, you can see the name is consistent across all of these records. And that's because this is a history of the task that we have here. You can see, for example, it was on m2 three times. That was my three times of bad luck in a row. And then it was w3, and now it's m3. So this is just the task history. So name really could have been dropped here. In other cases, if you had scaled up your application, you could have multiple slots, if you will, which is having multiple tasks to run multiple instances of your container. And so then the name starts to matter at that point in time and maybe you bring it back. So treat this as you need to focus in on what you need. We've got what we like here. I'm going to put the watch command back. Wrap that in a watch, and I put little single quotes around this. For some reason, I'm having an issue getting the format to work when I pass it to the watch command. We've got our table, and the nice thing is that table will update as we make changes.

Docker Container Kill to Change Placement Too!

So just to be clear, we have a similar representation of data here. On the bottom, we have this little box to represent our task, but it's only the last instance of our task. Up above, we actually have the prior instances of it in a tabular data format. So, when we come over here and do something at the command line a little bit different to trigger placement changes, what do you think you'll see up above here? Okay, so think about that while we work on this secondary approach to triggering placement changes, which triggers another question. Can you think of any scenario that would lead to a placement change besides the forced update we did? So the one thing I like to do is just to cause havoc. I rebooted my VMs, and that caused the task to be moved. Even easier than a reboot, I can switch context here, hop over to m3, and what do you think will happen when I run this command, a docker container ps and list everything? Actually, I shouldn't need to list everything. We just want to see the running containers. We can see we have our container right here for our visualizer service. If you want to match things up, on the end of the container's name, there's this u62 da, da, da, da. That is the task ID, and it should match the task ID up here on the left, and it does. Once we have all that in place, we know we are targeting the right container, we could do a docker container and a stop here. Maybe even more aggressively, we could do a kill, and we can target that viz. Use tab completion, be lazy here, run that, and let's see what happens. Hey, hey, do you see that up above? Can see we, hey, we got lucky. The very first time we did something to cause havoc, it triggered the managers to place the task on the w2 node. And of course, we could attempt to check that in the browser. The only problem is we've got our broken state now, which is great to see. And I also rebooted the VM, so my other container is non ‑ operational. For now, let's address what happened here to our service and see if we can't fix it so that it's placed on one of the manager nodes that it seems to need.

Docker Service Update --constraint-add node.role==manager

So, what exactly is amiss here? Well there's a couple ways we could go about this. I could pop open the developer tools and take a look at the Network tab. And if I click down here, it's not always quite so obvious. I think this is kind of a poorly ‑ designed message. It really should surface in the UI. But this is just a learning tool, it's not a production ‑ worthy tool, so it's fine. And hey, if this happens, it's not going to be hard to figure out that you're going to have to move the container and change the service definition to get it to work. So, what it's telling us here, it's kind of hard to see, I'm going to open up a new tab with this particular request. And this should be a bit easier to read. This node is not a swarm manager, and worker nodes can't be used to view or modify the cluster state. So this highlights an important piece of information, and that's that managers keep the cluster state, they change the cluster state, they preserve the cluster state. And none of that state is available to workers other than the work that they are performing. And there are obvious security reasons for that. We need to specify that our service can only be placed on a manager. To do that, let's hop back to our service definition. There we go. Now right now this is in the form of a docker service create. Fortunately, we've seen that there is an update command, so we know that we don't have to remove this to fix it. We could do that if we'd like to. It is a stateless service, so it wouldn't hurt anything. But let's take a look at what it's like to make just a quick change to this service. So, what we're looking for is to constrain the placement of our service. So we're going to do an update, and we need to specify what it is we are updating. So I'll give the name of our service here. And then I'm going to come up a line here so we can add the argument to make the change that we need to make for placement. Can you think of any resources we could use to figure out what we need to type here? Just like with docker service create, there is a great reference, so to is a good reference with docker service update. You can just google for that with ref, and it will typically pull you to the right place in the docs for Docker, under Reference, docker service update. And this also happens to be a really good page to scroll through and read at some point in time. It's not quite as long as docker service create, but it's got some additional helpful information in here. And if I scroll up to the top, I can get back to the arguments. And if I search for placement, there is a preference for placement. I don't want just a preference though. Aha, here we go, there is a constraint as well for placement. And so in this case we'll be adding a constraint. When you're dealing with updates, you are oftentimes talking about adding or removing a piece of configuration in terms of the service's desired state. Remember, services are a specification or a desired state. They describe what we would like to see happen, so we need to add or remove to that description. Now in this case, we are dealing with a situation where we probably aren't going to have the extra information about what constraint we need inside of this document. Can you think of a different one we could use? So, there is that docker service create command, and it is very verbose. It has a lot more than this one has. Inside of here, we can probably find what we need. So I'm going to search for the word placement. And aha, here we go. Here's a nice section down below. So always look for sections down below if you're not certain what to provide in terms of configuration, and see if you can't find the section you need. Now this is placement preferences again, so that's not quite what we need. Glancing around, I can see, okay so actually, over on the right ‑ hand side here we have service constraints. That's what we're looking for. And so another way to talk about this is a constraint. We're constraining the placement, it's making it a requirement versus a preference. And so now in this case, you can see we have various different attributes that we can target. For example, right here, we can see node.role as manager. That's exactly what we need. So I can copy that, I might glance down below just to look at how this works, so ‑‑ constraint ‑ add, and then place in the expression. And that should be good to go then.

Docker Service Inspect : A First Look at Spec, PreviousSpec, and Convergence

Okay, so all we need to do is lift our little call here, it's much more abbreviated than the initial one, and we can hop back over to the terminal and this time paste our command in. And when I run this, what do you think will happen up above? Let's find out. All right, looks like we've got a triggered placement change, a rescheduling of our task onto a different node, in this case m2. Now that could have just been random luck. One hint that things are okay is the fact that our application should be operational soon. This still doesn't mean we didn't get lucky, so I'm going to wait here for that to happen, and we'll come back and talk about testing this out. Okay, things are running now. Our service has converged. That means that it's reached the desired state that we had requested. In our case, we wanted a running application, that's the desired state here, and it is now running. You can see that once the current state hit running here, as it was ticking along, it was updating the current state in our output here. We're attached to the service's convergence to watch that process happen. You can detach if you'd like, but you're attached by default. And so now that our service has converged, that means we should be able to refresh this window, and hey, we've got nodes. All right. Should be able to refresh this window, and we've got a running application. So those are all good things, but this doesn't guarantee anything. Maybe we just got lucky. So a couple of things we can do because we could continue to push a button and trigger replacement forever and never really be certain that what we're doing is working. One other thing we could do, and I'm going to give us a whole window for this because there'll be a lot of information. We can run a docker service and then inspect, and we can take a look at our viz service. There's a lot of information that's going to come back. We're not going to go through most of this, but somewhere up in the middle you should see a previous specification, and then above that you should see a Spec, or specification. This is the current specification. So the last two states are maintained. So we could compare this to see if our constraint was added, meaning it was parsed. It seems to be correct. It's been added here. Let's make sure it's not down below. Okay, so where was that at? Let's make sure that was under Resources, RestartPolicy, and then Placement. So let's see if we get that down here. Under PreviousSpec, there's Resources, there's Placement, and there is nothing about a constraint. So, looks like our constraint was successfully added. If we wanted to, we could diff these two separate fields. This gives us a certain degree of certainty that we added a constraint anyways, and it seems to be working. Of course, if we'd like, we can test out and make sure this works a second time by forcing another update. Okay, it moved, it stayed on the same node, and so we could do this over and over and over again, and this would be our other degree of certainty. But the nice thing is we can see in our specification for our service that it's got that constraint in place. So that means it's going to stick there unless somebody takes it altogether off or deletes the service. And it is this placement constraint, this whole entire specification, the current spec, this used to derive the state of the cluster, the desired state. And so the Swarm manager nodes will make sure that desired state is enforced. For example, if a node goes offline that has containers running on it that were hosting services, it'll be the responsibility of the managers to make sure those exact same services are deployed elsewhere, in this case, onto a manager node.

Challenge: Deploy Your Favorite Web Service ... I'll Use NGINX

All right, so our visualizer works well at this point. We have it to confirm the state of our cluster. Now I want to throw a challenge your way. We've done a lot of poker and prodding with our visualizer service. Pick some other application that you know pretty well how to use the Docker image for it, something like a web app, perhaps NGINX or Apache, and see if you can get said application deployed onto the cluster using the service create approach that we've taken thus far. Okay, let's go through one of these together. So I've got the NGINX page opened up here on Docker Hub. If I scroll down here, somewhere in the middle should be your typical example of running the application and exposing the port. So this should suffice to remind us of the characteristics that we should care about. I'll blow that up a little bit, and then let's pull up a text editor. Alrighty. On the bottom, I can type out my scripts. So I start out with my docker service and my create, by the way, Tab+9 is great at suggestive things after you get to using it for a while. Hopefully it doesn't suggest anything inappropriate. All right, and then we'll come down here, and after our docker service create, what do we need to do next? Up above here, we've got a port that we need to add, why don't we put that in? So I'm lazy and I copied that over from the other service, which means we need to change the port, or we're going to have trouble. So how about we go with like 8200, okay, 8200:8080, ah, it's port 80 inside the container, which I love when people actually host their application on port 80 inside the container. Port 80 was meant for web, it is nice to see web applications actually running on that port. Okay, and looking up at our command here, so we name the container, well, that means we need a name for our service. Instead we have a similar syntax here, so name, and then how about we just call this weby. Okay, and then we have a ‑ d for basically a detach up here as well to run as a background container. We could detach and not wait for convergence if we'd like. For that, we just specify the detach flag. All right, now, what do we have left here? Looks like we just have to specify the image, that is a name suggestive a proprietary build of NGINX. We don't have that, so right on the end here, we can just type out nginx. And then before I forget it, since I'm wrapping lines here, just for readability and maximizing breakability, get those back slashes on the end of each line, all right, good to go there. So let's yoink this. And over in a terminal, let's go ahead and paste this in here, and see if we got it right. Cross our fingers. All right, we got an ID back for our service, so that is good news. If we get some blurb of random characters, all's good. But why didn't it update up above? That's a trick question. We're monitoring the viz service up here. If I Ctrl+C to cancel that, what I need to type in here to switch over to monitoring the NGINX service, so just the one change I need to make is to specify, hey, docker service ps, and then all that gobbledygook for formatting, and we want weby, and there you go. We've got our one instance of our weby service up and running, and it's been going for about a minute now. And you can see we didn't have any sort of convergence message down here at the bottom, that's because we chose to detach when we were creating it.

Docker Service Logs - Accessible Even for Containers Running on Other Nodes!

Okay, so a couple of things we could do. First off, it might be nice just to see the interface to the application. At the same time that we do that, I want to show you another really neat feature, and that is that you can actually take a look at the log files from any manager node in the cluster, and that's a pretty powerful thing. Because you think about it, the container might be running anywhere, and we're connected to m3 right now, so if we want to see the logs, those logs had to be shipped off to us to be made available via m3 somehow. That's no easy feat. Do you want to take a guess how we get access to the logs? So as always, we do our docker service, and if you look here you'll see in the output we have an option, or a sub ‑ subcommand, to take a look at the logs for our service. So we just type in logs here, and then we need to provide something here. What do I type in? So, weby. And there is our log output. If something went wrong with the service starting up, that information could be here as well. And saying that makes me want to check and see what's going on with our viz service. So we can check different services while we're at it, and unfortunately it doesn't have anything about the problem that we had when we had deployed to the wrong node.

Task History Defaults to 5, Log Following, and Other Log Output Options

Now by default, this is showing us everything available. The first thing I want to observe is that there are five entries in here, at least five repeating items. And if you look here at the front, we have, what's this? Well, it's a reference to our container, and it has the task ID within it, so it's also technically a reference to the task. So there's one of them. Here's a second one, a third, a fourth, and a fifth. There's exactly five. And I wanted to confirm that because you might've also noticed that under docker service and then when we were getting the ps for the tasks, put in viz here, you might've noticed there are five items here as well. Kind of hard to read with all the extra information, four and five. Why are there five and five? That just coincidence? No, it actually is keeping five items. So that's the default is to keep the task history for five tasks. And it's kind of odd in terms of where to go to configure this. You might think it could be on the service level. When you are setting up your swarm, that docker swarm command, when you initialize a swarm, there are a bunch of different options that you can pass, and one of those is the ‑‑ task ‑ history ‑ limit. And if you notice on the right ‑ hand side, the default here is 5. So that's where that comes from in terms of how many tasks back you'll keep per service and specifically per replica or slot within a service. We'll talk more about replicas shortly. Okay, and the reason I wanted to show logs originally, I kind of got distracted, and that's okay, there's some good things to see there, there are some options you can pass, and one of those is to follow the logs, so follow the log output. Tail is also an option to look at the last number of lines. So what I'll do is I'll run this again, put in ‑‑ follow here, and we need to provide weby. If you look at the bottom of the screen, you'll notice that I have not got back my prompt yet, and that's not just like an accident. If I hit Return, it still doesn't come back. That's because I am actually listening here for any subsequent output.

Swarm Networking Magic - Accessing Published Ports from Any Node!

So let's keep that log output open while we come over to the browser here and load up our NGINX application. What do I need to type in here to access that NGINX app? So this is where a bit of the magic comes in. All you have to do is hit one of the nodes in the cluster. You don't have to care which node you hit. So we've used 102 before. Let's use that one again. And then, do remember the port? It's 8200. Hey, hey, look at that. We have got our application, and that is pretty darn cool. Cool thing is I can come up here and change this to 3, and it loads. And hey, look in the bottom. You can see the log output from requests coming in, which is what I was hoping for. There's 4. Actually, there is no 4. We don't have a fourth manager. We do have a worker 4 though. Look at that. It loaded in the case of a worker. Let's go to worker 2, 112. So from an external perspective, if you're curious about how you get to the services running on your cluster, outside of your cluster, it's really easy. You can hit any of the nodes, and if you have a published port, and so the word publish in here, then that is going to be available to all of the nodes externally on that port.

Removing Services and Monitoring Removal with Swarm Scoped Events

I'd like to take a look at one more approach of how we can go about monitoring our cluster from the command line, and that's using the docker events command. So if you just run this on a manager node, there you get access to what we're going to see as both swarm and local events. So you can see some events have already popped up here, I'm going to clear these out, we'll talk about those in a minute. What I'd like to do those split the screen and trigger one of our own events. And right now, if I take a look at my services, look at the weby service, what do you think I'd do to nuke that service, and what do you think will happen up in the top when I do that? So dockers service rm, and then to trigger this, I'll need to specify which service I want to get rid of, and I'll get rid of weby in this case. We should have some events show up with regards to our service being removed. So this is a little bit jumbled because of the wrapping, that's one thing I don't really recommend when looking at the events, so however you need to do it to get rid of the wrapping, I can use the less command. And then I'm going to come back to the events command here, I'm going to go back in time since maybe 2 minutes ago. So now you can see the log output wraps off the side of the screen here, and then in the middle, we can see a high ‑ level overview of the events that happened. In this case, we have some of those exec commands that have happened. What these are health checks. So the viz service, the visualizer, has health checks built in, and this is just the status of those visualizers, and I believe that's because the visualizer is running on this particular node. And it looks like 2 minutes back is not quite far enough, so I'll go back maybe 5 minutes at this point. There we go, that's what I want right here in the middle. Here is our service removal. Another way we can target a specific type of event is to come into the event command, and in addition to since, we can add a filter, and then if you Tab complete, I find that's the best way to get to the various filters you might want to add. Also helps with the syntax here. So you can Tab complete through here, and what we're looking for is the type of event, and for the type, if I hit Tab again, I know that I'm looking at service ‑ level events, so I'll put that in. When I do that, you can see now we just have the service event coming back. Another way I could slice and dice that, instead of a type filter, I could go with a scope filter. If I go back here, I can show you all those filters again. So we have scope, pretty much we have a filter on any of the resource dimensions we might be interested in. And for that scope filter again, I can't get any completion here, but I know that I have local and swarm available, and it looks like 5 minutes back now is not long enough. Let's go back 9 minutes, there we go. We've got our service removal. So these will be all of our swarm events in the last 9 minutes. And that's the neat thing, if you really want to go back in time, you can. I want to say maybe 109 minutes, you're going to see everything I've done and any sort of preparation for any demonstrations here for the last hour ‑ and ‑ a ‑ half or so. For example, the node update here, that's because my cluster was offline, as well as the network update. We'll talk about networking shortly. These are just things coming back online when I turned on my cluster, my machine had gone to sleep, and so I had to boot everything back up with Vagrant. This log ‑ like output, it's pretty helpful, but it's a lot of information, and it's easy for it to wrap off the screen, you have to deal with that issue, or maybe you have to zoom way, way, way, way out , and try and get all the information on screen, which sometimes might work depending on the type of event that takes place.

Readable Docker Event History with JSON and jq

Like all the other listing and tailing commands where we're looking at output in some sort of scrolling format, there's usually a format option available, and it usually takes some sort of Go template. So, like before, we can set this template up using JSON, take the entire context and print it out per object, and we get this nice JSON output. Unfortunately, it is not formatted. We could do something about that, though. I like to pass this to the jq command. Not only will it format and indent the JSON, so it's very readable, where there's more or less one key ‑ value pair per line, if you're versed at using jq, you can use it to filter the output and change the output however you'd like. So the nice thing is if I go back in time a little bit further here, let's go back 30 minutes, and go back and drop the filter, there we go, we get our container events and everything else. So we have a lot of rich information to take a look at if you're curious about what's all available. What a great way to get an idea of what's possible by just going back in time. And this is where you might want to filter on different types of events. For example, exec\_create, exec\_start. Those are both the beginning and ending of health checks. So, there's an exec command run to get into the container. Once it's done, there should be an exec\_die. So after the health check is completed, it should have information back, and this represents that information coming back. Now, this is not always in a model where you get a bunch of details about the event. You do get the exit code, in this case, which is nice. You don't always get all of the details, though. For example, when we removed our service, we didn't get anything about the tasks that were removed or the containers that were removed or the process of that happening, so some events are more rich than others probably because a lot of these swarm events have been added more recently than the other events. In fact, if you want to see what's all available, if you come out to the Docker events docks and scroll down to the bottom, you'll start to see examples, such as node events, the secret event type, config, and then here's service. So for service, we'll only see creation and removal. For plugins, we'll see pulling and enabling those. So you can go through here and see what's all available, and some of the events have a lot more information available about them. These might be good just for watching what's going on and learning, or even in a production environment, if you want to monitor things that are taking place, then you can go react and query the state of the cluster to find out a detailed picture of things if any of these events don't have all the details you want.

Deploying Applications with Stacks

A Stack to Deploy Our weby NGINX Service

In the last module, we took a look at this new concept called a service that helps us deploy containers onto a cluster of machines, specifically a swarm. In this module, we're going to take a look at the next step, if you will, in the evolution of how we deploy our applications onto a cluster by taking a look at stacks. And if you have a minute, maybe you take a second, pause the video, and think about exactly what a stack might represent that you might be familiar with already. At the end of the last module we had two services, viz and weby. We nuked weby to take a look at Docker events and take a look at the process of removing a service. So, why don't we bring weby back this time with a stack? Over in Visual Studio Code, a couple of things. I created an examples folder, and inside of there I have a services folder. I set it up and copied over these scripts to create various services from the last module. For this module, open up the stacks folder. Inside of here, I have various examples, and open up specifically the weby set of examples. There are three files in here. We have weby as a container, as a service, and we have weby as a stack. Does this stack file look familiar? This should remind you of a Compose file, specifically a Docker Compose file, because verbatim it would work inside of Docker Compose to deploy as a standalone project or application, and it's also going to work here as a stack. And that's the important similarity with stacks that's going to help you propel forward and probably learn to love these a lot. Before you know it, you'll be abandoning that service create command because it's not exactly fun to type arguments out, a lot like with docker container run versus docker compose. So first thing first, let's go over to weby ‑ container, and let's compare it to weby ‑ service just to see how similar these two files are. Okay, and I know if I go into the Settings here, look for diff and put in space, I'm going to check this box here to ignore leading and trailing whitespace, and you can see the diff looks like there's even more in common . Other than that though, we only have two words different. So now let's just take the service from docker service create on the right side, and let's compare it to the weby ‑ stack. It's not going to be an apples for apples comparison. There's enough different that the lines are all colored, but the important part is that any dark colored area, so dark red or dark green, for example, with 8200 mapped to 80 here, same thing over here, that darker shade indicates that text matches and is on a line that has text that doesn't match. For example, the ‑‑ publish doesn't, and so there's a lighter shade of red here. If you look at what's in a light red color, we can see some of the overlap. We're creating a service here, and we're mapping this port. Now we've got the name and image flipped around in the two examples. So what I could do is just temporarily come over and move down the image so that it comes after, and now you can see all of our arguments, so what we're creating, a service, with the name of weby with port 8200 mapped into 80 and then with the NGINX image. Beyond these parameters, if you will, the rest is just scaffolding to execute our request. In one case we use docker service create, and in the other case we use a stack, which at the end of the day actually ends up creating a service as well. As you can see, we've got a services block here. So it's amazing how much still matches even though we're switching from a shell script on the left to YAML on the right. By the way, I linked into a Compose spec schema doc, and that's why you saw the hover over there that said compose ‑ spec a moment ago. This part up here is absolutely not necessary. It's optional. It's to do with the built ‑ in Red Hat plugin I'm using in VS Code in terms of the YAML language server to identify the schema for the document I'm working on. So ignoring that and ignoring the shebang on the left ‑ hand side, I'd say these two files should look familiar if you've never even worked with Docker Compose

Deploying the weby Stack with Docker Stack Deploy

Okay, so when we go about deploying our stack, what do you think is going to be different here at the command line? Well, first up, we had before the docker service entry point, some management command with a bunch of subcommands to help us manage one ‑ off services that we're creating and updating, removing, etc. in the last module. In this module, we're talking about stacks instead. And, accordingly, there is a nice docker stack management command that has a series of subcommands as well. Some of them looked very familiar. We have ls and ps, for example. So we could just run this one, docker stack ls. You'll see we don't have any stacks. Obviously, we haven't deployed any. And as for docker stack ps, we'll need to have a stack deployed to be able to look at its tasks. So I'm sitting inside of the weby folder, and there's my stack file. To deploy this, all I have to do is docker stack and then deploy, specify the stack file with the ‑ c arg. And then I need to name this as well, so we'll call this the weby ‑ stack. Immediately we are presented with creating a network and a service. So okay, a network, that's odd. We didn't ask for that, I don't think. But just like with Docker Compose, there are implicit features, and networking is one of those implicit features, and that's why we're getting the weby\_default network. Can even look at the docker network ls, and you can see we have our weby\_default network created, and this time, interestingly, it's actually an overlay driver. If you've used Docker Compose before, you would've seen a bridge driver on your user ‑ defined network. This is overlay because we're dealing with a cluster of machines, you can see the scope is swarm, so it has to extend beyond just bridging devices together on a single device. So that's how our container gets connectivity. But the most important thing is our service was deployed, so let's take a look at the docker stack ls command again. You can see we have our weby stack up and running with one service. We can take a look now at the docker stack and then ps if we'd like, dump the tasks for our weby stack, all of them, and there is the one task that represents the single container that's running as a part of our weby stack. If I want to open this up in a browser, what do you think I need to type in to the URL bar? So, just like before, I need to put in my IP address of any of the nodes in the cluster, and then we're running on port 8200 as the port that we published. And there's our NGINX service. So it's up and running. That's pretty cool. If we want to look at a few more things, we could look at the docker service ls command, and you can see we have both our viz and this time we have weby\_weby for our service. And that's because, like with Docker Compose projects, we prefix the stack name, so weby was the name we chose for the stack, in front of the various resources inside of that stack, which contains our weby service. And of course, just wanted in the last module, if we could go about looking at our tasks this way, but we'd have to provide weby\_weby. That's why tab completion is very nice to have set up. And there's the same information as above when we use docker stack ps. So, a lot of different ways to get at the same information to see what's going on. At the end of the day, I hope you find it rather convenient to work with a file in terms of passing arguments so you're not copying and pasting commands and you're not worried about typing out arguments by hand. A file can be checked into version control. It's a nice thing to keep safely tucked away with a nice change history as well.

Updating a Deployed Service - Just Edit the Stack File and Push the "Deploy Button" Again!

Now for the real power of a stack file, as well as the real power of a Compose file. Let's say we want to change something about our deployed weby service that's a part of our weby ‑ stack. For example, how about we go about changing the port that we're published to? Let's switch from 8200 to 8300. What do you think we do to set that up? So all we have to do is edit this file right in place. Put 8300 right here. Forget about the old value. Save the file. Hop right over to the command line. I might run a quick docker service ls so we can look at the port that we're running on. You can see we're using 8200 right now for our weby\_weby service. Now what do you think I'd do to deploy the changes that I made to that file? Well, I just have to run the deploy command again. I'll fire that off using the same name for our stack. Keep in mind you can create separate instances of the same stack file, so you'll want to keep the names the same unless you're intending for a separate instance. And that's the neat thing about using stack files, almost like a function that you can inject with new arguments and get a new instance out of, a template, if you will. All right, you see it says Updating service weby\_weby? Let's run our docker service ls, but do you want to guess what's going to show in the output here for the ports? Well, hey, look at that. We've got 8300 now instead of 8200. But the real test is to switch over to our browser, make sure that 8200 is broken. That's great. Always love it when we're looking for things to be broken. And 8300 is working. It was an incredibly easy way to go about making a change to a deployed application to change the port that we're running on. It's not always an easy thing to do. That required getting rid of the other published port and then adding in a new published port. It also required inspecting the current state of things to see if we had any published ports or not and then how to get rid of them and then how to add the one we need. It's all a lot wrapped up for us inside of the logic inside of Docker Swarm, specifically the swarmkit logic, which is where this resides. Takes care of handling a lot of crazy scenarios for us so we can just edit a text file at the end of the day, push a button, and we're done. It's a very convenient workflow. So if you're familiar with Docker Compose at all, this will feel right at home. It's even more powerful than Docker Compose though because you actually have a cluster with the reconciliation process running for you. You don't have to run Docker Compose to get the reconciliation to happen again. It's happening all the time for you. And the best way to understand what I just said is to take a look at an example of changing the nodes on our cluster such that the container that we have running can no longer be running where it's at because maybe we pulled the plug on a node and see what happens then to our weby container here that's a part of our weby ‑ stack.

How to Monitor All Tasks at the CLI

This next example is going to entail modifying one of our nodes such that one of our tasks for a service has to move elsewhere. And I'm just looking at this and thinking, well, we're working with the weby service right now. Why don't we do something to m2 so that weby has to move away? And I was thinking there's actually a practical use case here. I'm glad that weby landed on m2. That use case is that manager nodes tend to need to be only manager nodes, especially as you grow your cluster. They need to save the resources for the task of either being the leader or essentially the follower and replicating the state of the cluster in the case of some sort of failover scenario so they can step up and be the leader. So let's say that we want to change m2 to fulfill that task, to give all of our resources on m2 to the management task. Well, we can do what's called draining a node to effectively make a manager node only a manager node because by default they can be both a manager and a worker. Now to do that, I've set up a script that we can use. This will help us monitor things as we make changes. It's this display ‑ nodes\_tasks inside of the host folder inside of the autos folder. Now this is not with our weby ‑ stack, and that's because it needs to be easily accessible from our Docker host. And because I'm using a Vagrant VM, it just so happens to be there is a shared folder set up that's rooted at this Vagrantfile, and so this host folder that's inside of there will give us easy access to these other files inside of that shared folder as well. We'll see that in a minute. So this display ‑ nodes\_tasks, inside of here I have two commands, actually three. I've got a docker node ls up above to list out our nodes, and then I've got a custom format string on here. Then down below I've got an all tasks, and this prints out information about, well, all tasks, and it does that with docker service ps, and embedded inside of it is a call to docker service ls ‑ q. And docker service ls ‑ q, we probably haven't seen that yet, so let's take a look at that. What do you think that does if I'm embedding it in that other call? Let's run it and find out. We run this, this is two separate line items, and that represents the two separate services that we have. You can see their IDs match up. The nice thing is if you print out just the IDs, those can be fed into the docker service ps command, which we know requires one or more services. So we're feeding in all of our service IDs and getting all of our service tasks back then. You can see what that looks like by running the script that's in vagrant/host/display ‑ nodes\_tasks. And there you go. You can see all the information that's printed out up above. Now I've got the screen split so that we can run commands down below because what I want to do is watch this script up above so that I can notice what the changes are as we make changes. So this is another way to monitor the status of things, is to build up a little display script, if you will, with whatever you want inside of it and then watch that display script. Okay, so I have the command saved here. Notice we are just passing to watch that exact same path that I just ran individually. So every half of a second we'll execute that, and that's why I'm sitting right now on my Docker host that's inside of a Vagrant VM. I want to have the performance to be able to crank it up to this interval. Outside of your Docker host on a client machine, probably want to turn this to 1 to 2 seconds, depending on how long it takes this script to run. All right, so I run that, and now this is updating every half of a second, roughly.

Understanding Reconciliation by Draining a Manager Node

Okay, now down in the bottom panel, I can run commands, and we can watch what happens up above. So I'm going to quick make a new tab so that we can take a look at the docker node command. So we have a couple of commands that at first might look like what we need. There's a promote, and there's a demote. And we could run this on the manager, but then we're making the manager just a worker if we were to demote it, or in the case of a worker, if we were to promote it, we would be making it a manager, then. We don't want that. Instead, we want this update subcommand, so let's grab some help for that, and there we go. We've got what we need here. We're looking for availability. You can see over on the right ‑ hand side, we have three choices for availability, active, pause, and drain. Active is where we're at right now. Pause would be don't add any new work to me. Drain means don't give me any new work and get rid of the work that I have. So that's the one we want. Note that we can also update the role here if we'd like. So the demote promote are just syntactic sugar over the top of this update command. Okay, so let's close this tab, come back here, and we can do a docker node, then, we can do an update, set our availability, and mark it as drained, and that's why I also like tab completion, by the way , down below. So let's go ahead and run this, and just watch what happens up above for both the nodes and the tasks. And specifically, we need to hone in on m2 before I forget that, so we need to put that on the end. Watch the line for m2 right here, and then also, watch and see what happens with our weby service, which is running on m2 right now. Alright, our node is drained, and then you can see we've got tasks changing on the bottom. And then, up above, we've got to change to our m2 node. So you can see we are now in the drain availability. Down below, you can see that our task was moved from m2 to w4, and you can see there's a little counter running here. This is just a side effect of the fact that we're looking at the diff and watching this command, which prints out the duration since something happened if it was recent. Most importantly, we are now running on w4, w4 our weby task. So let's go confirm that in our browser. There you go. We've got our weby task moved over here to the w4 node. I should say the task slot because the task itself ended its life on m2 when we set things to drain, and a new task was created over here on w4. So tasks are a one ‑ shot deal. They don't actually move around. However, the slot, which is the idea that I have, x replicas, so I have x lots. And in this case, I don't care where they're at, but they have to be somewhere. That's what moves around. And finally, we get a new container over here. And, of course, the nice thing is if I go to a browser and refresh the site for the application that's a part of the service that we just pushed around, for the end user, there is nothing to update. I don't have to change which node I go to access that particular service, even though it's moved around on my cluster. It's a very convenient thing. And that's a feature of what's called the ingress swarm load balancer. So as traffic comes into your cluster, you can hit any node. It'll be load balanced to wherever the service is running at. So, when I say that there's a reconciliation process constantly running to make sure that the desired state is enforced, that's what I mean by making a change to the node, m2, which triggered a change to our service task. Now if we were manually orchestrating and deploying containers, we would have had to have spun down the container and spun it up somewhere else, and then we'd have to do that for every container running on m2. And if moving those containers caused any problems like too much load on a given machine, we might have to move other containers around to fit things in if we have a container that's doing a lot of work on m2. And, of course, at that point, we wouldn't have managers either because we would be the manager, and that's why this is so incredibly powerful. It's like your Docker Compose, specifically your stack file, just imagine it's loaded up into memory, and it's constantly being pulled to make sure that the state in your stack file matches the state on your actual cluster.

SwarmKit Has One of the Best Ever Documented Codebases

I want to share one resource that's really helpful. So there's a lot of documentation for Docker out inside of the Docker docs. We've seen some of this. Sometimes I find, though, the explanations aren't sufficient for what I'm looking for, or I find them vague, and I would like a little more explanation. One of the best resources I found is the SwarmKit codebase itself. I think this is one of the best documented, in terms of internal documentation, codebases I've ever seen. I want to show you that briefly. So if you come out here and just go through the files on GitHub alone, that is probably good enough. You can clone it down to your machine too, but you've got everything out here, and GitHub is pretty easy to get around these days. So if you hit T on your keyboard, you can search for files, and then .proto is one of the number one recommendations I have for looking for help in terms of explanation because this is what's used to describe a lot of the interfaces that you and I interact with at the end of the day, I should say the Docker engine interacts with, but that we interact with through the Docker engine, so the APIs defined here. specifically, look inside of the API folder. There are also definitions for the API between the worker and the manager nodes, how work is doled out. And so if you go through here as you need to learn about things, sometimes there's a better explanation. A couple of spots that I find really helpful, number one of which is the specs spot. I've got this opened up already in a tab here. So you can see this is actually part of the API that we interact with because it's defined as the container objects for user ‑ provided inputs so that would be from the Docker engine. So let's say I'm curious about what exactly drain means. I could just type in here drain, and because this is an API that we interact with, it's likely going to have the definition that I want. So we've got our Availability enumeration here in this node spec. So that looks exactly like what we're looking for. We've got Active. We've got Pause. We've got Drain. So those are the exact options we saw in the command line. And then above here, we've got these nice little explanations. Drain nodes are paused, and any task already running on them will be evicted. So that already opens the door to maybe looking for some other terms like, What does evict mean? Or what does task mean? Or we could look for task inside of here if we wanted to. Task defines the task template. This service will spawn. It's pretty cool. So this is on the service spec. So, hey, here's a definition of a service. So one thing I'll tell you, a key piece of information, when you're running in spec commands for SwarmKit functionality, like with weby\_weby getting the service inspect here, whatever you see inside of here is stuff that you can look for over inside of that .proto file or somewhere else inside of that codebase. So if I was curious about UpdateStatus, we can look and see if that's inside of here. It might not be a hit every time, but I could also come over to the command line. I've got a copy of SwarmKit pulled down here. I could run through and search for UpdateStatus and see what I come up with. So I'm getting a lot of hits here. I've even seen some documentation with some of these hits. Here we go. UpdateStatus contains the status of an update, if one is in. And it looks like I'm truncated a line here. This is in the objects.proto file, so I can open that up. I was interested in line 127, so here we go. Okay, if one is in progress. So this objects.proto file contains definitions of things that are on that service spec that we got back from our Docker service inspect. There's our UpdateStatus. Here's what an endpoint is. Another helpful dimension to this documentation is the code itself.

SwarmKit's Code Is Itself Accessible - Check It Out!

I found that it's not just the documentation in here. I actually think that the code itself is extraordinarily readable. For example, here is the worker.go file, so here's what a worker is. So this particular worker implementation takes care of core task management, which, of course, is the whole job of workers to take tasks and execute them. So if you want to learn how that works, this might be one of the files to look at. So here's part of it, read the tasks from the database and start any task managers that may be needed. So there's some notion of a task manager. If I wanted to get a little bit deeper, I could go look for that and start to understand the implementation of the various features of SwarmKit. One of my favorites, though, and I've got this opened up out here on the website, here is the dispatcher, and it's an API provided by a manager group for agents to connect to. So agents connect up to this dispatcher, they receive their task assignments this way, and then they report status this way. Types is another favorite of mine. For example, if you have a task in a given state and you don't know what that means like pending, for example, well, here's the definition of pending, waiting for a scheduling decision. So at that point, we know our work hasn't even been scheduled yet. And so then we have to ask, well, what might be causing our task to be pending, not scheduled? One case might just be there's no resources available because there's some constraint on it that doesn't allow it to be placed anywhere on the current set of machines that we have, so we've overconstrained it. And then my last place to go, one of my absolute favorites, is the scheduler file, and this is the logic for actually scheduling work. If you really want to understand the reconciliation side, how the managing happens for you, the orchestration, all the tasks you would have to do if you wanted to place machines yourself or place containers manually, all that logic is in here to understand, and I think it's easy to understand because I think you can understand the concept of SSHing into machines and starting containers in various spots if you're to try and manually stitch together a cluster. So use this as a resource as well. As you have time and as you use SwarmKit more and more, don't feel afraid to get into the code base itself and at least start with the definitions and the little comments here and there. Here, for example, is the scheduler event loop, so this is a core piece of functionality. This just runs over, and over, and over again, performing the same work to make sure that the cluster has converged to its desired state.

The SwarmKit Repo Design Docs Are Filled with Architecture and Terms

Another helpful section in the Swarmkit repo is the design folder. Inside of here, you'll find a series of markdown files that go over terminology, and in other cases, explanations of functionality. For example, you can learn about raft and how it's used and implemented in these two files. It's used by the manager nodes as a protocol to communicate cluster state to replicate it and ensure that it's always preserved or perhaps a little more high level if you are curious about the task model, how exactly work. Well, it's a really nice detailed explanation here that's not terribly long, it explains each of the pieces of state within a task, what its purpose is, even gets into the task lifecycle, task history, which we've talked a little bit about , and then way down below is the slot model and that's what I've been talking about when I refer to a tasks slot, that's the .1.2 etc depends on how many replicas you have for a task. There is also a nice nomenclature document. I found that the terms inside of here are concise and often insightful in pointing to other terms that will help understand them. For example, just in looking at agent here, we can see agent coordinates the dispatch, so we saw the dispatcher earlier in the code of work for a worker. So the agent maintains a connection to the dispatcher and then not only do tasks come from the dispatcher, but the agent turns around and tells the dispatcher what's exactly going on with a given task so it provides state back to the dispatcher so that that can be aggregated on a cluster level and that's how we get access to that information when we're running our command ‑ line queries. Or we can see, for example, that a task is running, when it's started to run, and that gives you the computation of how long it's been running or if there is an error, a failure, what happened.

Creating a Stack File to Deploy Our viz Service

Alright, for this next example, I want to turn our attention to a different scenario. I want to look again at our viz ‑ service. So before, we were creating this with the docker service create command, now I'd like to look at what it looks like as a stack. If you'd like a challenge, you can pause the video and try to convert this to a stack yourself and see what you come up with, and then continue on with the rest of us. Let's walk through this. I don't really consider the stack file to be the important part. It's a good exercise, but it's a rather trivial stack file. I've got it created here already. Let's just walk through this quickly. We have our image. We have our ports. We have volumes. Now this is a bit simpler, I think, than over at the command line with docker service create where we need to use the longer mount syntax. Over in the stack file, we can just use the traditional shorthand, if you will, volumes syntax because we're just mounting in the docker socket. That said, that is a plus one for the stack file approach because I think this is easier to read. Then let's split the screen here. We've got our service create up on top because I want to talk about the placement constraint. That's really the novel thing here as far as stacks are concerned. This is a feature that you wouldn't have found inside of Docker Compose, and that's the deploy section or element inside of your config file nested underneath your service. In this case, I'm calling it web. So underneath of deploy, we have placement. I appreciate how this reads. It's deploy with a placement constraint to make sure that the node's role is manager. I think it reads really nicely just stepping through things there, even for someone that's new to a stack file, as opposed to the command line approach where we're just passing a constraint argument, and you might wonder, well, what exactly are we constraining? And, of course, there's quite a few other aspects that can be nested underneath this deploy element, and that's where you're going to find most of what's novel about a stack file in terms of the configuration of the service. So there's nothing too fancy here because we already went through the scenario of dealing with a constraint that was missing. Remember, we didn't have the node role manager to begin with, and that causes some problems when our task wound up on a worker node where it didn't obviously have access to the swarm manager dataset. So we've dealt with that constraint that was missing. Now we've got another constraint, and this one I think is really cool because it's an implicit constraint, constraints that aren't so obvious from the actual configuration. It's not that there's anything missing in the configuration. I want you to see how powerful it is to just change this configuration file slightly and how Docker Swarm reacts to that and adapts to a dynamic environment to successfully deploy the application after first failing to.

Initializing a Pi Swarm in under 3 Minutes Using a Bare Metal Setup of 5 Raspberry PIs!

So let's talk a little bit about the environment first. If I pull up my list of contexts, you'll be able to see right now I'm connected to pi1, and I have pi1 through pi5, and that's because I've set up a Pi cluster for us to use and all the nodes have Docker installed on them. I wanted us to hit the ground running, but I did leave one thing out, I didn't initialize the swarm because I thought it would be a good opportunity to practice something that we tried at the beginning of the course, really just to see how easy and powerful it is to set it up in a brand ‑ new environment. So I can show you that the environment is not set up. I've got a rather long command here. You have to trust me on this. This basically loops over using Z shell syntax five numbers which leads to me calling docker and info on each of our various Pi nodes. So we've got pi1, pi2, pi3, pi4, pi5 using the context arg here, and I'm just asking to get the information out by formatting with JSON to just grab the swarm info only and formatting it nicely with jq. So let's take a look at that. You can see all the nodes here in a second will come up as inactive. All right, all five of those are marked as inactive, so we have a good starting point to initialize the swarm. So I'm connected up to pi1. Why don't we just use this as our very first node? Do you remember what I type here to create a swarm on the very first node? So that is the docker swarm init command, and then do you remember what we need to pass to this? So we need to pass the advertised address, the address that the managers will use to talk to each other. If I want to get that, I can just use a quick call to SSH. I also have SSH set up to each of these nodes. So it's pi1.lan and then I can run a command like grab the IPv4 address, address show, and I can look through here and I know that this is the address that I want here. It's this 2.101. So now I can come back and I can run my docker swarm init and I can pass my advertised address and paste in that 2.101. When I run that I'm connected to the context of p1. Bam, we've got a cluster created just like that. I can do a docker node ls and there will be one node here. I'm always impressed with how fast it is to create a secure environment because these nodes are actually using encrypted communications out of the box from the very starting point, and that's what those tokens have to do with. They're cryptographic tokens that help secure the communications, as well as secure new nodes as they're joining. So you'll want to protect those tokens as well. Now we have some output here that gives us the command to run to connect up a worker. I am just going to clear this out though because I want to start by connecting up two and three as managers as well. And to do that, I will need the join token. Let's just pretend I forgot to write that down or didn't query that in the very first starting point of initializing a node. Do you remember how I can go about getting that? So I need to be connected to the context for one of the existing managers. I can do a docker swarm and then join ‑ token, and then I just need to specify a manager or worker, in this case I want the manager token. I guess while I'm at it I could grab the worker token too. So that's how you can get access to those if you forget what they are. So let's just go ahead and copy and paste the manager token, the entire line here, it's the entire call to docker swarm join ‑‑ token, and let's just paste that into the console here. And I'm going to come back to the start here and I'm going to change the context to pi2 because I have all the context set up to point at these various nodes. So by doing that, I'm going to tell pi2 to go ahead and join as a manager, run that, and there you go. And then I can come back here and I can run it for pi3, and there you go. We've got three managers added to our swarm cluster. Docker node ls if we're curious to see, and we've got all three nodes. And now I want to create the last two as workers, so I'll grab the worker token join ‑‑ token statement for that, paste that in, go over to the start, change the context on this to pi4, and then we' ll change this to run for pi5 as well. Just like that, we have our entire cluster set up. Isn't that pretty impressive? And another reason I wanted to show this was because when we first started out I wanted to explain some conceptual foundations so it wasn't just me throwing this at you. Now I'm kind of throwing at you as a refresher, as well as an opportunity to see how easy it can be to set up a cluster. Okay, let's take that stack file and deploy it to our cluster.

Troubleshooting a Failed Service on the Task Level with `Docker Stack ps --no-trunc`

Okay, so our nodes are ready and I'm sitting inside of the example folder with the viz files, notably the viz ‑ stack on the end. The context is also set up to use pi1, so I am ready to go to deploy my visualization stack to my newly created Pi cluster. Do you remember how we deploy a stack? Just type in docker stack deploy, and then we need to specify the file for our stack, so viz ‑ stack in this case. And then we also need to give the stack a name, so we'll call this viz as well. Okay, that is deployed, and you can see that our service and networks were created. Do you remember how we look at our stack? Well from the command line we had the docker stack command, we can list out our stacks. And there you go, we've got our one viz stack, it seems to be working, at least it was recognized by the manager. How do we dig in and see what's really going on though? Well, as far as our services are concerned, we look at those on a little bit lower level. So we can go down to the service level, take a look at those, and you can see we have our viz\_web. And so the \_web comes in the fact that the stack file, I use the name web for the service, so it's tacking that onto the end of the stack name that I chose. So there is our service, and at least as far as this screen is concerned it doesn't look like anything's wrong except that we have 0 of 1 replicas, and that is a problem. So we're running what's called a replicated service type. It's scalable to any number that we specify that our cluster can handle. And right now, we only have one instance running and it's not operational yet. Now perhaps that was just a delay in the process of grabbing the image and starting things up, so we could always grab our service status again to see what's going on. You can see that we still do not have an instance of our service up and running yet, so that would be a task and specifically a container. So something is probably not right here. How can we take a look at the task level for our stack? There are two ways. We could go down to docker service and then ps, but don't forget we have docker stack and ps as well. This one just requires the name of the stack, and it will print out all the tasks for that stack across all the services. If we go down to the service level, we'll only get the tasks for the service. Now we might want that if we're only having a problem with one service of many in a given stack. Okay, so we have some more information here. This is where I'll give you a little tip on the docker stack ps command, and many of the ps or list commands in Docker, there is a no ‑ trunc option, which stands for no truncation. If we run that, we'll get more details about what's wrong, as well as more verbose detail about our digest on our image, which we probably don't really care about, as well as the full ID, which we also don' I don't care about. What we do care about though is some of the details about what's going wrong, which you can see here, we have no suitable node, unsupported platform on 3 nodes, scheduling constraints not satisfied on 2 nodes. What do you think this part means, the scheduling constraints not satisfied on 2 nodes? Well, remember, with our stack file we have a constraint of the role being manager, and thus two of the nodes are workers and, therefore, don't suffice as the node that this can run on. Then, as far as the other three are concerned, what do you think might be wrong with those? Well it says unsupported platform, and that's specifically hinting at something. The image that we are trying to run on these various nodes that are managers, it was compiled for a different CPU architecture than Pi nodes have. Pi nodes have an ARM processor. If we want to confirm that we can grab a docker system info, dump this out here, it'll give us a client side, as well as a server side, but we're only concerned about the server side here. So I have my client up above, just ignore that. Down here on the server side, there should be an OSType and an Architecture, so you can see we need armv7L, basically armv7 is what we're looking for as far as Docker images are concerned.

Finding an Image's Supported Platforms on Docker Hub or via `hub-tool tag ls --platforms`

What exactly does the image have available? Well, a couple of ways to get to that piece of information. So if we hop out to the browser, probably the easiest way to go is to come to Docker Hub and you could explore to the image or you can input /r/ and then go out and copy the slug for the image, paste that into the browser, and that'll pull up the repository. This is the official build of this example as far as dockersamples, that organization out on Docker Hub is concerned. Underneath Tags then, we can come and look at the latest tag. And if we look at this, we can see that the OS architecture is set to AMD64, so that's not going to work quite so well with an ARM processor. If you're interested in a command line approach to get the same platform information, one of my favorite ways to do that is with the new experimental hub ‑ tool CLI. If we get the help for this command, we can see there are a series of subcommands, and by the way, you will need to authenticate to use the tag command that we're going to use in a moment. One of the subcommands to this command, and by the way, I believe this Hub CLI tool will eventually find its way into the Docker CLI. It's just experimental at this point in time, and it comes with Docker Desktop by default, so you have it if you have Docker Desktop. Anyways, there is a subcommand called tag, which we want to take a look at, and underneath that are a series of sub ‑ subcommands, number one of which is ls. This will help us list out the tags for a given repository, so then what we can do is provide that same dockersamples/visualizer. And there you go. We have the two separate tags listed, but we don't have anything about the CPU architecture. So, what we can do, I'll split out a new tab here, we can grab the help for this command, and underneath here you can see there are some options, one of which is to include the platforms. So I can come tack that onto the options for the ls command, and when I do that, you can see on the right ‑ hand side we have the architecture listed as AMD64 as well.

Changing Images to weshigbee/swarmgs2-viz: Just Edit the Stack File and Deploy!

So what I did to make this work, I went ahead and created my own image that does have support for multiple platforms. I'll probably try to submit this after my course, since I just forked the swarm ‑ visualizer repository that exists, and if I don't, I'm sure somebody will. So if you actually see that the doctorsamples visualizer has support for multiple platforms in the future, don't be confused, it's just at the time of recording it doesn't. I mention that because we ran all those commands at the command line and looked at the website, and we could see that it only had AMD 64 support, and that could change. This one is located here at weshigbee/swarmgs2 ‑ viz. If you look at the tags here, you'll see right here I have different architectures supported. I really like drilling in here, though, because you get a nice drop ‑ down to see in big print which of the architectures are supported. And, of course, from the command line we could also look at the same information, we could paste it in the repository where I have these images at, and you'll see the difference here, over on the OS architecture side we've got linux/arm/v7 listed in addition to linux/amd64, so this could run both on my Mac or on these Raspberry Pi's. Interesting, on my Mac with Docker Desktop, I can run any of these platforms, thanks to \_\_\_\_\_ support in Docker Desktop. If you're really curious about cross platform support , as far as pulling down images and running them with a different architecture, you could look at this article, and I assume that you should be able to get this set up and running with the Raspberry Pi's as well, so that you could run the AMD 64 image and emulate it on the Pi's. So keep that in mind, too, if you're trying to simulate this problem on your local machine and you're using Docker Desktop for Mac or Windows, well, it's got cross ‑ platform support ready to go and set up, whereas with the Linux Docker edition that is out on my Raspberry Pi's, it does not, and that's why I can encounter the problem there. Okay, back to the demo, because I want to leave the assumption in place that I can only run arm images, because I really like this implicit constraint that comes along and bites me, and by implicit I mean it's not listed directly explicitly in our stack files. Its indirect by virtue of the image that we referred to having support for a given set of platforms or not. So what do we need to do then to get things fixed so that our stack works? Well, we just need to hop over to our stack file and change out the image. So I can go ahead and paste my image right over the top for my repository that'll use the latest tag which has cross ‑ platform support. Now what do I do? This is one of my favorite answers. It's going to be another docker stack deploy. Just repeat the same exact deploy statement we had before, keeping our stack name of viz the same and our stack file the same. I should note I did save the stack file by the way. If you have any problems with things, it might be that you didn't save your stack file. Alright, and it looks like it's updating our service viz\_web. Now for the drum roll, let's see what happens here, docker stack ps, look at our viz stack here. So this is interesting, and I'm glad I caught this before things changed, because you can see right now the desired state is set to Running, and the current state, on the very end it wraps around there, is set to Preparing still . That's often the case when you're pulling an image that takes a little bit of time. So let's clear out and let's run that again, docker stack ps and viz, and now you can see our current state has converged with our desired state. By the way, the hour ago, that's because I forgot to add this into the original recording, so now I'm editing it and I'm adding it in. It's important to see that the current and desired states match, and I really like the fact that you could see that convergence in the original output not quite having converged yet, which you can oftentimes see when you're pulling an image for the first time onto your cluster or even pulling it onto one node in your cluster for the first time. Let's quick confirm what we see at the command line by switching over to a browser and pulling up the visualizer web application. So, I just need to navigate to 2.101, the address for the first Pi, port 8080, and there's my Visualizer. I also have a name set up here, pi1.lan, just to conveniently access these Raspberry Pi's. Most notably, though, I want to switch to another Pi and look at port 8080, but before I do that I want you to tell me what's going to show here. So I'm pointed at pi2.lan now, port 8080. Well, we'll see the Visualizer. We'll see it on every single node, even one of our worker nodes, so pi4. That is the ingress swarm load balancer at work, same thing we had before working on our Raspberry Pi's. These are literally five Raspberry Pi's now. I've got 8 GB of RAM on them, the platform is armv7, I've got 1, 2, 3 managers, and then I've got 1, 2 workers. So, as you can see here, a bare metal installation of multiple machines is another great way to learn about Docker Swarm. All I had to do was install Docker, you watched me do the Swarm in it in a matter of minutes, and I even deployed a stack file with a service in a few more minutes, reluctantly with a problem too, and troubleshooting it and everything within a very short span of time. It's a very convenient platform to work with when it comes to clustering together multiple machines. And then, even more so, when you think about the fact that you use a Compose ‑ like format, in fact the exact same Compose format, with some additions for deployment purposes, to just throw containers at this cluster and not really care where they land, as long as they land somewhere where they can run. It's as if we glued together five separate machines and get all the power in one, and then even better, because we're using Docker we have the ability to then slice that power up on a container level as if it were just a bunch of processes running on one machine that we can isolate completely from each other. Even if there's multiple containers working together in one stack across multiple nodes, we can keep them isolated from another stack running on the same nodes or different nodes.

Configuring a Scaled Service via a Stack File - echo-replica.yml

For this next set of examples, I'm going to use my running VMs locally, and then I want to go over to the examples/stacks folder and open up the echo example stack. Inside of this echo folder, there are two separate YAML files. We have global, and we have replica. We'll start out with the replica file first. So I've got the file opened up in my editor here. Let's talk about the specifics. I've got an image here. This is an image that I built with some endpoints inside of it for learning purposes, one of which is the echo endpoint, if you will. I just intended to more or less echo back request information so you can kind of start to understand how a request comes from you, gets into the cluster, and then gets routed to an actual container somewhere to be handled, and then how it comes back . So seeing what kind of information I could gather along the way and then sending that information back so we can compare it to what we might expect it to be versus what it actually is. We're publishing port 9090 that maps to the web app here on port 80. I'm mapping in the Docker socket. I have some examples where I read some data from the Docker API much like the Visualizer uses it to gather information to show. Then underneath the deploy element, I have mode set to replicated, and this is redundant. It's not necessary because it wanted's the default, but I you to see it explicitly listed here because this will be the one big differentiator versus the global version of this service. And then I've also set a number of replicas to 2. So when we very first create the stack, we will have how many containers on our cluster? Well, 2 exactly because we want 2 replicas.

Using a Templated Task to Append {{.Node.Hostname}}-{{.Task.Slot}} To Service Container Hostnames

And the last thing I'm doing here is using a template for the hostname for the container, setting it to echo ‑ replica and then ‑ , and then I'm injecting values here, the node's hostname and then the Task.Slot after that . You'll see why this is helpful later on. If you pop open the File Explorer, there's a README file inside of which I have a link to a little bit more information when it comes to what exactly placeholders you have available to inject inside of templates and where templates are applicable. So we can use them in a hostname, in a mount, and an environment variable. And then these are the various different placeholders we have, and they're rather telling about what they inject. And then down below, you'll have an example of doing this with docker service create and setting the hostname here as well. These are all deferred until the point at which the task and the container are created, at which point they are interpolated and injected with the actual values of where that task is sitting at. And so the nice side effect of this is that our task will have a name that includes its node's hostname so we'll know where it's running at.

Deploying a Stack Based on weshigbee/swarmgs2-echo

All right, so how do you think we go about deploying this replica stack? We just need a docker stack deploy, pass in the stack file, put in replica there, and then, of course, we need to name this. Let's call this echor. That's all we have to do then. Fire that off. If we want, we can hop over to a browser and take a look at the visualizer, and you can see already we have two replicas that have been created. They're not running yet. That's what the red dot means. Probably they're pulling an image right now. You can see the state was preparing at the bottom and then starting and now running. So they're both up and running now. So they had to pull the image down. Unless, of course. they already had the image on them, then they would start up a lot faster. So, we can see we ended up being placed on worker 2 and worker 4. Random luck. So that's what scaling does as well is it increases the number of instances of containers that we have available to handle requests that come into that service. In this case, for that 9090 published port, so why don't we look at that? So I've popped open a new tab, prefilled the URL to 99,101, and we can hit 9090. And there you go. We've hit our new echo service. This is just one of several endpoints available in this image that I like to think of as a diagnostic image for learning a little bit more about Docker.

echo's /containers Endpoint to List Its Docker Host's Containers - Akin to docker container ps -a

Another endpoint useful is the containers endpoint. This happens to read information about that particular Docker host. The output is very similar to the docker container ps command with the all flags, so stop containers would be shown as well. That's why I mapped in the Docker socket. It reads the containers that are running there. So you can see, on w2, it looks like we had an old version that exited of viz, so this exited 11 hours ago, and we do have 1 running task right now for our echo service. And of course, that makes sense. There has to be a container running on this host to answer our request. Literally, this is the container that sends us back what is essentially a docker container ps for that particular host, including the information about itself that would show up in docker container ps. Now the convenient thing, because I use that template, you can see w2 up here inside of the host name, because that's the host name of the actual application that's running inside of that container, and thanks to me setting that to the node host name , we can see which node this is running on, as well as we can see the slot is number 2 here, so this is the second of the 2 containers. Now, I can probably refresh this over, and over, and over, and I'll continue to get w2 ‑ 2. Let's talk about why that is, next.

Forcing the Browser to Establish New Connections to Investigate Routing Implications

Okay, so I mentioned that I can refresh this page probably until I'm blue in the face, and it probably won't change which of the two containers is handling my request, even though there are two of them that could possibly handle the request. That's not always going to hold. But the thing with browsers, they tend to hold onto a connection using keep ‑ alive, so they keep a connection open to make it easier to make subsequent requests for the same or similar resources on the server. And thus, the connection information doesn't change either. Over in this view over here you can see every once in a while perhaps the port changes. And if you're really lucky, you might see the IP address change, indicating that you switched the container that's handling your request. It's not very often though, and it's not very often if you're refreshing this connection that you're going to see this happen. The timeouts only happen after a period of inactivity or if you force them to happen. So I've got another set of windows opened up with Chrome here, ready to go to show you what I mean. And the first thing I want to point out is that you can see here the IP address for the local IP, which would be the server that's handling the request. It's different. It's not 10.0.0.28. It's 10.0.0.27, so this is our other container handling our request. And it just so happens we were lucky to get that one, and it's different, but it's going to stay the same. Just like with our other browser, until you're blue in the face, you probably won't get lucky to get that IP to change unless, of course, you force it to. So, in Chrome, there are some tools that you can use to close out idle sockets and to flush socket pools to basically break active connections, as it says here. It sounds it may break active connections, which great to me. So, what I've found is you can click this button here, and then come over here, and sometimes you'll get lucky. When that connection is broken and you refresh, you'll get a different IP address for the local IP, indicating that something else is handling your request, one of the other containers. But again, if you continue to refresh here, nothing's going to happen. And I found that with Chrome you can continue to refresh and just click this button over on the right ‑ hand side, and you'll see that sometimes it changes, and for some reason it doesn't change the context of the windows when I 'm clicking this button over here. So, if you're refreshing really quickly over on the left side, when you're clicking that button over on the right, it doesn't seem to steal focus. So it's a really great way just to see the values toggling back and forth as you flush out the socket pool. And you go 27, 28, 27, 28, 27, 28, duh, duh, duh. Most likely, what you're going to see happen is the remote port probably will change if the IP address doesn't change. That means that you do have a new connection coming into things. This connection ID will also change as well, but you won't see the remote IP change, and you will not see this IP change because we're not adjusting the browser's address bar. And then this is what is more or less a proxy that's taking our incoming request to 99.101 on port 9090 and load balancing it off to one of the containers that's available. It comes back through this address, then. That's why we can see that from the perspective of the handler, it's like we are 10.0.0.2, but that's not us. That's somebody else acting on our behalf, the ingress load balancer that handles incoming requests to your cluster and then routes them off to the appropriate location. In this case, including with load balancing and specifically with Linux, Docker is setting up what's called a virtual server using IPVS, so there is a virtual IP, and that is what's taking care of the load balancing for us. It's a kernel ‑ level functionality. And I don't need to get into the specifics of it. You can look up Netfilter and iptables if you'd like to learn more, and specifically, IPVS. And don't get me wrong, the specifics of how IPVS works is very fascinating. You should look into it if you're curious, even look into how it's implemented with Docker, but think big picture when it comes to Docker Swarm. This is yet another piece of the kernel that Docker, in this case Docker Swarm, is taking advantage of gluing together with all the other features that it uses for container isolation, restriction of resources, etc. into this concept of not only just a container, but this virtual concept of an overlay network with service discovery that's load balanced, in this case. Okay, popping the stack back out of that discussion into where we are at, the important takeaway is that if you're working with the browser, keep ‑ alive will prevent a change in who is handling the request when you have a scaled service because the connection stays open.

Deploy to Pull the Latest Image

On the flip side, if you come over to the command line, the values are extraordinarily garbled up, and they are changing, but I realized once I put some nice formatting on these pages that I forgot I wanted to actually look at this from the command line. So I've actually updated the image to take out the table formatting on a few of these endpoints inside of this little echo image, so that we have textual output that we can look at on the command line and not go, what, with all this HTML tags. So what I want to do with that is run an update and get that new image deployed. All I've done is push out a new image on the latest tag. Now typically you probably would have an updated tag. You go into your stack file and edit the tag and then deploy things. We don't have that because I'm just updating the same tag, so what we can do then, well, you tell me. What could we do to trigger the new image to be pulled and our containers to be recreated with it? Well, first up, I need to make sure I'm sitting next to my stack files, and then all I've got to do is run my docker stack deploy again, specifying the exact same values, not changing a thing inside of that stack files. Give that a second to run, come over to the browser, and I can refresh here, and eventually you're going to see an updated version of the application. There you go. Can you see that change over right there? So now our request is being handled by 10.0.0.29, and if we flush out our sockets, there you go, 10.0.0.30 is the other one handling our requests. So things have been updated. Oh, by the way, look, we've got w3 right here, which is new. It was on w2 and w4. And let's see if we can get the other container and its node to show here. There we go, w1 and w3. Okay, so let's go over to our Visualizer and confirm that. And there you go, you can see that is indeed the case. On w1 and w3 we have our two tasks or containers now, pulling the latest image that I just pushed out that goes back to a textual format for a few of the endpoints that I wanted to use from the command line.

Observe Routing with Curl and without Connection Reuse

Because what I want to do is come over here to the command line and run curl again. Same request we're making in the browser, and that looks a lot easier to understand. To see the benefit of keepalive differences with curl, what do you think I'm going to do here at the command line? Let's pull up that curl call. Let's put a watch on here. Let's run this with differencing and let's run it every 1 second. I'll drop off the title to save some space. Now we should probably have curl not print so much information, so I'll put the silent flag on here, as well as capital S to show errors if they come up. Okay, that's much better. So now we've got the similar output as before, and you can see every time a request is made here we're getting a new IP address almost every single time. We've got a new remote port, we've got a new connection ID, which means there's no keepalive in place and the connections are being created every single time a call is made. So this, I believe, is a little bit more interesting to look at because you can see things changing as the requests go out and are routed to the various different service handlers, which are our containers, and we have two of them right now. So they're just going to alternate back and forth, perhaps every once in a while you'll see the same IP address show a times in a row, but things are going to be load balanced between the two. You can even see there in the hostname that you have w3, w1 changing as well. This 10.0.0 network is your ingress network. It can be customized, you'd want to look that up in the docs if you have any conflicts with using this particular IP address range. And it's for handling all the traffic where you have requests coming in for published ports going off to containers then that might be anywhere on the cluster. I want to point out one other piece of information printed out here. Down at the bottom is a list of interfaces. Those are interfaces coming back from the server or container that responded to the request. Just if you're curious, you can see the various networks then that that container is connected up to. For example, the 10.0.0 network and, of course, its IP address is 29, that matches the 29 we see up above under the local IP that's responding to the request. We can also see the 10.0.6 network. And there's also a 172 network. The 172 network, that's involved in routing traffic out to the internet. It's a bridge network that's installed as a part of setting up a swarm, so when you initialize a swarm. You'll see a docker\_gwbridge if you're to list your networks, and that's just providing NAT'd external connectivity, and only out to the outside world, not to other containers. And, of course, a loopback interface. All of these interfaces remind me how easy it is to take for granted everything Docker sets up for us. This is crazy complex networking right here just for one container, let alone for all of them. It's truly nice to be able to sweep all this configuration under the rug, declaratively describe the networks that we need, or implicitly in some cases, and just benefit from that simple configuration.

The Role of the Ingress Overlay Network

I want to take a moment and do a bit of a recap and set us up for a nice segue into the next module about jobs. So we've got our vagrant cluster of VMs that we were just working with on the most recent examples. I'm going to organize things somewhat linearly here so that I can talk a little bit about networking and convey some of the more advanced aspects. Otherwise, I'm afraid this will become somewhat confusing. So in the last couple of examples, we looked at scenarios where, from the outside world, whatever that might be, it might just be another computer on your network or it might be somewhere across the internet, somebody has sent a request in to one of your nodes. Your nodes have IP addresses, they can contact those, and they've reached out to some published port on those nodes, in our case, published ports for this echor\_echo service for the echor stack, basically just that echo image, and of course, 9090 is available across the cluster thanks to setting up that published port, and it maps into port 80 on echo. Now for the examples I've done, I've always hit the same node, but it doesn't really matter. You could have hit any node. Let's spend some time to show you that as you make requests, they'll be randomly routed to different backends behind the scene and that's why we saw the IP address changing in the browser when we force a connection to reset, and especially with curl at the command line where we had a new connection each time. Our requests for being loaded balanced across the various different containers that we're running for our service thanks to the swarm load balancer. Sometimes this is also referred to as the routing mesh. You might even hear the word ingress and that's important because what we've somewhat glossed over is that communication has been happening over what is known as the ingress network.

Communicating Internally over Default and Custom Overlay Networks

Now I've dropped one of these connections off on the left ‑ hand side here, so I want to pretend like we're talking about one connection in. And I want to ask the question, what if, when we make this request, we need our container that's handling the request to make some subsequent requests to other services? We've talked about this as the service discovery issue. For example, it could actually make a request to itself running on a different node. Where exactly would that traffic get routed? Well, there are many possible answers to that question, all of which revolves around the fact that there are other networks involved besides the ingress network. The ingress network is meant for published ports. It's not meant for general communications between containers, and therefore, why when we create a stack, we create networks. In our case, when we created the echor stack, we created a default network, and that allows communication, then, between the containers that are spun up on the echor stack. This network was implicit, a lot like Docker Compose creates implicit networks for you for your project, so you didn't see it anywhere defined inside of our Compose file. Now to better understand this network, I've got an example that won't just connect to one other container. Let's say we spin up instances of our containers on all of our nodes. We can look at a global service type where one container is literally run on each node, as opposed to providing a number of replicas that are spread across the cluster. And the example I have is going to send a request from one of the containers to all of the rest of the containers on the stack. So this is happening behind the scenes across that default echor network, and the response is coming back to us through that published port, thanks to the ingress network. And it is important to keep in mind this is just a simplified example, so I only have to run one container image. Inside of it, it has a special request that can reach out to other containers. And I'll show you how we can plug in a domain such that it will reach out to all of its neighboring containers. Doesn't really matter, though. It could reach out to one container on its network or all of them. These examples will suffice to show that we have communications going on behind the scenes on other networks. We could even have yet another network, perhaps another application, and perhaps we can communicate across that network as well to some secret back ‑ end services or some sensitive back ‑ end services and make requests for protected data and include that as well. So it doesn't have to be other containers on the same stack. It could be other containers on a different stack. And that's some of what we're going to see as we move into this concept of a job. So with all that said, let's back up here to what this looks like for this request that I'm going to make across the cluster. Keep this visual in mind. Also, keep the visual in mind of a single container.

Performing a DNS Lookup from One Container to Its Own Service - and We Find a New, Implicit Stack Network!

Okay, so the very first demo I want to perform, and by the way, I've got a visualizer up here on the bottom with the state of my VM cluster. Interestingly, since my VM was rebooted last, both of the tasks for our echo service, they've both landed on the same node, and that's probably because N ‑ 1 booted up faster than anybody else, and then just took all the work. Anyway, keep in mind, that's where they're at right now. Up in the top here, I've got a request that's baked into this image as well. It's more or less like performing an nslookup or a dig request to take a domain name, so it's /dns/lookup, and I'll paste in this parameter here, domain=echor\_echo, that's our service. So what I'm doing here is I'm actually going into this container, controlling one handles it, either of them, and I'm asking it to do a lookup on a domain and give me back an IP address. What IP address do you see to talk to if you wanted to communicate with any one of the containers that's a part of this echo service? Do you want to take a guess what you might see when I run this? So we get something similar to the command line. I've truncated the output here just to keep it simple. I've made this request with a .NET application, and then I've just grabbed out the IP address that comes back, and I've shown the type of IP address is IPv4. Now, that's kind of interesting. It's 10.0.6.2. That looks a little bit different. Let's split the top screen here, and over on the right, I've got the request up that we were originally making when we were looking at the connection flopping back and forth between handlers, and we're seeing this IP address change here. Now the difference is this was on 10.0.0.11. Well, this happens to be our ingress network. So our request is going across the ingress network in this case, because we're reaching out to a published port, and it's being handled by one of our containers, and we can sometimes refresh. And look, you can see we have a change here to 10.0.0.32, still on the 10.0.0 network, as that flops between being responded to by these two separate containers running our echo service. Of course, I probably can refresh it for a long time here and never see a change again. So there are two separate networks involved, 10.0.0 and 10.0.6, that's what I wanted you to see out of this. I'll zoom in a little bit more if that helps. So when we communicate to our echo service, we use the ingress network from the publish port. When one of our containers for our stack wants to communicate with another container on the stack, it uses a different network, this 10.0.6 network. And that's exactly that separation we see here between the ingress, or 10.0.0 network, and what we'll see here as the ECHOR\_DEFAULT network at 10.0.6. Now, if you hop over to the command line, we can confirm these networks actually line up as I suggested. We can run a docker network ls, that suggests to be connected to one of the nodes where the manager is at and perhaps where one of your containers is running at, you'll have the best luck. Worker nodes don't always have access to the custom networks that might exist like our ECHOR\_DEFAULT network, unless of course there's a container on them that needs it. It's part of the efficiency of Swarm and making sure that it doesn't scale out the networks too large, especially as your cluster size grows. Anyway, ingress is what I care about right now. What I can do then is run a docker network inspect and pass along ingress, and I'll format that with jq, just to get some colors. Look way up at the top here. You'll see there's a Config section underneath which you have a Subnet set to 10.0.0.0, no surprise, /24. So there is our 10.0.0 network, and you can see the gateway on it is 10.0.01. Now if we're really curious, we can come over here and say, okay, so we have this remote IP; it's still 10.0.0.2, and these have changed, by the way, as my VMs were taken on and offline as I worked through these demos, but this IP, unfortunately, it hasn't changed. It could change too, but it hasn't, and that's nice; 10.0.0.2. We could look through here and see if we can find 10.0.0.2 anywhere, and there you go, we've got 10.0.0.2. This is literally listed as the ingress sbox or sandbox, so this is where our traffic is getting routed through. This is our proxy if you will, that's handing requests off then to the various different containers behind the scenes, and we can only see those because I was so fortunate as to capture that and send it back in the request here in the echo request. Otherwise we'd have no knowledge of this IP address. It actually would look like everything was probably handled mostly by this particular request host. The 10.0.0 IPs would be obscured from us. Now, there's a bit of information missing from this view. Join me in the next video for that.

Docker Network Inspect's --verbose Flag Is Filled with Rich Details About the Ingress Network

So while I've been digging into this ingress network by inspecting it, I found the 10.0.0.2 what looks like to be a proxy or load balancer to hand the request off to one of the services containers and toggle back and forth as new requests come in. But I don't have a lot of information about what exactly this 10.0.0.2 is and where is it at and that's because there is actually a hidden flag here that you might miss otherwise, and to show you this flag, take a look at the stacks folder, go into the stress folder or you can actually just look at the contents of it from here, and look inside of the notes folder inside of there. I have this diff verbose network inspect, I want to run that so I'm just going to path to it, but before I run it, I'm going to print it out for you so you can see what this does. So we're going to run something very similar to what we just did with the docker network inspect and ingress network, except we're going to diff it with the docker network inspect and then we're going to add a verbose flag, and again, the same ingress network. Now this is the label I'm showing you that will show up at the top of this diff. These are the actual commands that are exactly the same here, and through process substitution here, we're passing this off to the icdiff command, and I'm asking to see the whole file, don't just show what's changed. So we can go ahead and run this then and this is going to show us what it looks like when we add the verbose flag and I wanted to show it side by side. Okay. I think that's good there. Now I know this is going to be a little bit hard to read, I'll read you through the contents. I just want you to see that up above here, we have pretty much the exact same output. The yellow is the same. So everything is the same up until we get to this point where there is a services field, and underneath of here, we have extra information that we wouldn't otherwise see. Now I know it's hard to see, but do you see anything interesting underneath this services property or field, Well, down under the IP addresses for the various different ingress endpoints that look a lot like what we just looked at, we have 10.0.0.2 again, in fact, let's look for 10.0.0.2 here. There is the original instance that we found, it's in both outputs, but then in the verbose output, we get an extra entry and this one has a little info attached to it, an info field that has a host IP, huh, it's set to 99,101. So it looks like this 10.0.0.2 load balancer or proxy, whatever you want to call it, is located in some sort of ingress endpoint sandbox, some sort of container ‑ ish concept that's hidden behind the scenes located on 99.101 and that's probably because we 've been making our requests thus far through 99.101. So over here on the side, if you got 99.101, we're getting 10.0.0.2. What happens if we go to 102, instead of 101, which is manager 2? Well, we could probably predict this. Let's come down here and look for 102. Okay, this is that same part we would have seen on both sides. Here is the extra bit of information. Okay, so 99.102 maps to what looks like 10.0.0.3 as opposed to the 10.0.0.2 on 99.101 up here. So if my math is correct, we should get back 10.0.0.3, instead of 10.0.0.2. Now let's find out if that's the case. So watch right here. I'm going to run the request with 102 and look at that, what do you know. We've got 10.0.0.3 just as suspect. Compared with this verbose flag, you can actually dig into some of the details behind the scenes and I don't have all the time in the world to explain all of this. I just wanted you to see that if you go spelunking through the network inspect, you're going to learn a lot and there is a lot to learn here that I can't cover in this course, but that I encourage you to look into on your own if you're curious about how networking works under the hood. So what it looks like is that each of our various nodes has some sort of load balancer sitting next to that port 9090 that's published, it's that little element that then load balances the requests off to the various backend service containers, and each published port is going to have its own little load balancer.

Investigating How the echor\_default Overlay Network Works for Internal Communication

All right, I want to briefly talk about the similar concept to the ingress network and load balancing with regards to our echor\_default network. How, after all, do we go from one of our containers to the other one? By simply providing a DNS lookup or just reaching out with curl and the name of our service. How do we do that and get to a response back from one of the containers that provides the service that we're interested in? After all, there could have been more than one of them. How do we know which one to go to? Why do we only get one IP address back here when I do a lookup on echor\_echo? It's 10.0.6.2, and it's always going to be that value, unless, of course, we change our service, maybe reboot our VMs, this value could change, but it's going to stay constant as long as our cluster is up and running, the same way that our remote IP over here stays constant as long as we're coming into the same node. So let's take a look quick at the command line at the same diff that we looked at of a network inspect with the verbose flag and without, except this time I'm sitting inside the notes folder, and in between recordings, I went ahead and create a second copy of that diff script, and this one is for the echor default network. And then I made the original one, so it's been renamed, and then if you look at the contents of these, I've got a diff set up here, just to show you there's not much difference between these files, I went ahead and parameterized the command down below to inject in a network in each of the various spots that's needed. This is like a function down here that shared where I copypasta ‑ ed it. So, let's go ahead and run that echor\_default script and look at the network inspect with and without the verbose flag. Okay, now glancing through here, you should see down at the bottom there's an extra section when we use the verbose flag. And by the way, the headers up here will show you which command was run. This is provided by that labels arg inside that script. So in this case, I'm adding the verbose flag. Well, something should look interesting here. We've got a VIP of 10.0.6.2. Well, VIP, that's what IPVS used as a virtual IP. So as you probably suspect, IPVS is not just used on the ingress network, it's used on our custom networks as well. So, 10.0.6.2, that's familiar because we saw that over here when we did a DNS lookup on our services name, echor\_echo. Remember, this is just our echo service prefixed with the stack name. We get 10.0.6.2 back. And that's a virtual IP. And it looks like it provides access to 2 separate IPs, 10.0.6.3 and 10.0.6.13. And right now, those are both relevant on 99.101, and that just happens to be because our service has 2 containers on the same m1 node. This is a lookup table, if you will, of how to go from the virtual IP to the actual what's called real service behind the scenes, or real server. If you really want to learn a lot, you can dig in a little bit more like we did with the ingress network, and we were able to make this request from the browser over here to get connection details, go right to that 9090 published port , but if I want to see what things look like from the perspective of one of these service containers, my echo service, I'm going to need to get into that container because I didn't provide any sort of web interface to this. So I've got a new window opened up here, I've already got a pre ‑populated Docker container exec, so I'm going to jump o,n inside of, my context is set to m1 because I need to have that so that I can get to the container, and then I already tab completed to get the echor service, one of the two, and then I'm going to run the bash program when I exec in or break into this container. I could have done this instead of using these interfaces right here, especially this one right here, because now I could do something like an nslookup, and we could put echor\_echo, and you can see we get back 10.0.6.2 as well, just like we have here in the browser. Or I could use dig if I wanted to, and I'll get the exact same result back. This is coming from the embedded DNS server provided by Docker as a part of these custom overlay networks, very similar to what we had with the custom networks with Docker Compose. Those have embedded DNS servers as well where you can just provide the name of whatever service you're working with and it will map it to whatever IP address you can use to communicate with it. And in this case, an overlay network, since we could have multiple instances of a service, this is a load balancer, or the front ‑ end to what will become a load balancer, that gets us back into the actual containers providing this service, one of which we're sitting inside of right now. So if we really want to have fun now, not only could we do a dig, we could do a curl. And I went ahead and preinstalled curl in here. It's not inside the image, but I added it with an appget update and an appget install curl. Now, what do you think we're going to see when we run this right here? So keep in mind, our echo service runs on port 80. We don't want to go to 9090 right now, we want to go to port 80 because we are inside of the network. We have access to the actual exposed ports of our application. When I run this, I'm going to get the same output that we had in the browser when we hit port 9090. We're just going to have different IP addresses because we're going to be using a different network to communicate. So I'm asking to talk to my service on port 80, so I'm inside of my custom echor default network. This maps to 10.0.6.2 as we saw, and then that gets mapped eventually to 10.0.6.14, which load balances off to 10.0.6.13, in this case, if we run this, ironically enough times. Look at that, 10.0.6.3, we're getting a response back from ourselves, and I can show you that here. Here we have an IP address 4 listing of what our addresses are on this machine, and you will see here is 10.0.6.3, which is what we got back the second time we curled that inspect or echo endpoint. If you're curious, you can come over and look at that diff, and now that 10.0.6.14, the intermediary, it looks like our load balancer right here. Where is that coming from? Let's come over to this window, 6.14, and hey, it shows up here on both sides. This looks like a load balancer. It has lb in front of it, and then our network name, echor\_default. So here's our load balancer for our network. So we have our VIP of 10.0.6.2, which is what resolves for the service name. That gets mapped off to a load balancer at 6.14, which then routes the request to one of the containers down here, 6.3 or 6.13. Now, there's a lot of rich information in here. I don't want to get into any more of this, but I just want to give you a glimpse at what's under the hood and help you start to pick apart and understand what's going on behind the scenes when it comes to these custom networks, as well as the ingress network. Because I really think it helps to understand, especially, that we have multiple networks involved. But that's all you walk away with. I think you're in a great position.

dig servicename Returns our Service's VIP - whereas - dig tasks.servicename Returns Our Service's dnsrr List of IPs

Okay, so a moment ago I said we had this simple request that we've been performing to reach out for that echo information or inspect information to look at who's handling our request. That's interesting, but I said there's another scenario where we could have any number of containers running beyond just two, and in our case, we'll get one running on every node. And then what happens if we would like to reach out to each one of these, perhaps to pull some diagnostic information? Is there some way we can go beyond that load balancer, which doesn't give us much choice about who handles our request, and get some sort of listing of the various services and decide who we're going to request information from? Well, yeah, we can absolutely do that. So over in my browser, instead of echor\_default, and I can do this here or I could go to the container itself and do this, probably do them both to show you, I can just prepend this request with tasks., so the domain will be tasks.echor\_echo. In this case, I'm saying, hey, give me a listing of all the tasks to provide this echor\_echo service. And now you can see we don't get 10.0.6.2 back. We get 10.0.6.3 and 13, which we know are the two IP addresses of the containers, see them down here, providing the echo service. So another way I could do that, I could come here and perform the exact same request, task.echor\_echo. And now, instead of 10.0.6.2, I get 10.0.6.13 and 10.0.6.3. This is accessing what's referred to as the DNS round ‑ robin approach. In fact, if I run this enough times, you should see the IP addresses flipping around here in the list. There you go. You can see that 10.0.6.3 and 10.0.6.13. That's the case because then you can oftentimes just pick the very first one that comes back if you use this approach, and you'll know you're not overwhelming the system by going to just one container every time because they're being rotated for You. They're being mixed up as you get the results back. So this is in contrast, and perhaps nslookup is a little bit cleaner here. There we go. So this is in contrast to using our load balancer here with our virtual IP or just going directly to the containers themselves. This is our DNS round ‑ robin approach, and this is our VIP, or virtual IP, approach with load balancing. Up above, we have to provide the load balancing, if you will, by picking one of the entries that comes back from the DNS response and hopefully not picking the same one every time. So sometimes people use this approach up above, and then they'll put their own load balancer in front of things if they don't want to use this built ‑ in VIP load balancer. Now in terms of resolving these various domain names and getting the IP address back, I want to point out the DNS server that's being referenced here. So this is provided by Docker as a function of these custom overlay networks. It's handling DNS for us behind the scenes, so we can just know these friendly names of services that we type into stack files, and it will take care of everything else for us to get us to either a VIP if it's load balanced, or if it's round ‑ robin, to give us a list of tasks that are handling requests for our service. I think that's pretty powerful. And so notice this is running on the current node that I'm on, and I know this is specifically set up by Docker because normally on my network I have my own DNS server, and it's not shown here.

Using dnsrr for Internal and Container to Container Service Discovery on Backend Overlay Networks

So now for the fun part. We have this containers endpoint that will return the containers of whatever randomly selected service container responds, so it would be all the containers on its host. And interesting right now, this list will be the same regardless because both containers are running on the same host and I'll show you that actually. I have another endpoint available. In front of containers, put cluster, and then on the end here, tack on a query string with a domain set to tasks.echor\_echo. So I built in a little piece of functionality that allows you to tell a similar containers endpoint to go query all of the responses that come back from this DNS request. So if, for example, we get two IP addresses back for this DNS request, both of those IPs will be contacted at the /containers endpoint to get a listing of containers that they have on their hosts and then those will be all coalesced together and brought back to this page with this new cluster/containers endpoint. So I'm going to submit this here and we'll see what it looks like with two containers. Now for a quick refresher. Here is that query for the DNS lookup of the tasks.echor\_echo and you can see it comes back with the two IP addresses for our two echo containers as opposed to if we just asked for the service, which we'll use the VIP and only give us the single IP address for load balancing, and thus for tasks.ecor\_echo, we get two separate hosts back, which happens to be the exact same host, in this case, because the two containers are on the same host. Before we move on, I want to show you how we can mix things up here and get rid of some of the history that might be showing up with a lot of changes to our cluster over the various examples. These lists might be really long for dead containers, for example, and if you'd like to clean some of that up and/or just want to mix it up so you can see more than just the m1 node, then let's take a look at scaling at the command line. So I'm going to run this here at docker service scale, and I'm going to scale our two prominent services viz and echor\_echo down to 0. You can actually scale multiple at a time, by the way. And as soon as that completes, I'm going to run the scale down again. I've had better luck with this clearing out some of the old state, but most importantly, this will really jostle up our containers and hopefully ensure that they land on different nodes, and then I'm going to turn around and scale these back ups. So the double scale down to 0 seems to have cleared things out in terms of the task history, give that a chance to start up, those are two echo services. And now if I do a docker service ps here on viz, we just have the one entry, and hopefully, we've got the same here with echor\_echo. We have two entries for the two running right now. Great. And the net effect here is that we don't have any of the old task history now. And if you look at the node column, you can see we now have two separate nodes for our echor service and 1 in w3. We could also open up the browser to see that things have moved around and confirm we have m1 and w3 for our echor\_echo service. Now I want to ask you a question. What will we get back when we resubmit this page now. Well here it goes. Is that what you were expecting? Things look a little bit simpler now. Now I do want to point out there is still one dead container listed here. This is the standalone visualizer that we had deployed. Obviously, we won't be clearing that out by dealing with the Docker Swarm interfaces because that was created with the standalone Docker interfaces. We could prune things on our various nodes if we wanted to clear out some of the standalone history as well. And another benefit of doing this, you can now see we have w3 as 1 of the nodes that we're working on and m1, so now you can see containers from these two separate hosts and they're different. So now just imagine if we had this available for all of our nodes, and to better understand this, just to make clear, when that request comes into one of the containers, right now we just have two, could have many of them, it then reaches out to the various other containers depending on that domain we provide and calls essentially the containers endpoint on those, coalesces the results back up and spits them back out, and that's what that cluster/containers endpoint does. Right. Now we have two running so why don't we get a full complement running, and to do that, we're going to look at a global service and that I want to use as the basis of segueing into our next module because we' re going to start talking about service modes or types, one of those is a global service, but then we also have two more that happened to be job types, a replicated job and a global job.

Running Jobs on the Cluster

Introducing Service Mode with a Quick Look at Global Services

So now we want to turn our attention to jobs. We've been looking at long ‑ running services, and we're going to wrap up here and segue by looking at a different type of long ‑ running service and see how that relates to other types of services or modes that relate to jobs. So right now we have the two instances of our echo service. We've seen how DNS round robin can help us reach out to every single instance of our echo service. We're going to use the same technique when it comes to jobs to deploy a job that does a web stress test of a service and uses all of the instances of that service, so you want to use the same DNS round robin approach. Now additionally, we'd like to see how this round robin approach plays out when we bring more tasks online, specifically we want one on every single node because then we'll be able to get a listing of all of our container history across our entire cluster with one single web view. That's something that's hard to get right now short of connecting to each node and performing a docker container ps. So we're going to have the setup with a global service for echo, and then we're going to reach out real quick and take a look at what that cluster's container endpoint looks like. And then we're going to use that to pivot into talking about jobs with an example of a job that does something very similar. So I've come out to the echo/stacks folder. Inside of here we had those two stacks, one was global and one was replica. I'm going to dump out the contents of the global one. And really, the primary difference here is that we just need to set the mode to global. In fact, I could run a diff here and we can look at those differences. And the primary difference is changing the mode. So I'm going to copy this because I want to stick with the same name that we have here so we can use the er even though it should be eg, just to simplify some of the commands we've already been working with; otherwise, I've got to say echo g\_echo and it's going to get confusing. So that global file is just for reference. Let's edit our replica file. So I'm sitting over here inside of Visual Studio Code, and we'll just replace mode: replicated and set it to mode: global, and then I need to nuke the replicas line because that's no longer applicable. And then over here at the command line I can take a quick look at my stacks. Those are echor stack. So that's what we're going to update. Do you want to tell me how we do that? We haven't done it in a while, but we just need to do a docker stack deploy, pass in our replica file, the one that we just edited, and let's keep the name exactly the same here. This is something I wanted to point out, and we'll get around it another way, but you cannot change the service mode once you've deployed a service. Well you tell me, what do you think we could do? We can just do a docker stack rm, get rid of our echor stack, and now we can go ahead and deploy it again and bring it back online. So that's our quick workaround if we can't change some value that we would like to change.

Switching a Drained Node to Active Triggers Reconciliation to Schedule Task(s) for Global Service(s) on the Now Active Node!

Now, do you want to guess what the visualizer look like before I switch to it? Here we are. Is this what you expected? So it looks like we have the echo service on the various nodes here, except we don't have it on m2. We can refresh here, and it doesn't look like that's something that's just updating. Why is that? Well, if you remember, we did a demo a while back where we drained the node m2, which means it's not going to be receiving any new work, and that's why it doesn't have an instance of our global service. Now, the neat thing about global services, if I were to bring that availability back online, so I can update that node, I need to set the availability here, and we'll set it to active, and we will target m2 here, make sure you do that from a manager node. You can see m2 is now Active. And what do you think we'll see here when we switch to the browser? Look at that. Now we've got it on all of our nodes. So if you want, you can leave it drained, or you could bring it back online. And now, for the grand finale, if I come over to the browser and refresh here, what do you think we're going to see? Let's find out here. So we've got tasks.echor\_echo as our domain. Look at that. Here is our container history, and hopefully nothing embarrasses me here, across our entire cluster. That's pretty neat to see and something that's pretty darn useful, if you think about it. And one thing I've never really pointed out, labels are what's used behind the scenes to help track which objects belong to a stack, which particular stack they belong to, for example, echor here. We'll zoom in and get over to the side here. So this is part of how Docker swarm mode goes about figuring out which objects belong to which stack. That way, if, for example, you edit your stack file and just change the name of a service, there's actually a prune command that can go out and nuke the old instances of it because it's kind of hard to tell if you change the name what exactly is going on. The stack file no longer has the old name in it, so how do we know what the old name was? Well, we have to keep it somewhere, and we've got it in labels here, so we could go query all the various containers that are running have a given stack namespace that matches the namespace of our stack, echor, and then if they are no longer applicable, we could just hack them off, create any new ones, and move on with our day. So this is part of the process of keeping track of what the current state is so that we can reconcile it with future desired states. Now, before we wrap up, I want to come back to the top here, and I've zoomed in so you can see, we've got each of our nodes. I want to go through as quickly. We have m2, we've got 2, m1, w1, so you notice, they're not in order, w4, m3, and w3. They're not in order. This is likely the order they came back from our DNS lookup. And if you were to refresh this page, probably would get a different order again. There you go, m2, m2 are the very first 22, and I do not recall those being in order. Refresh again, there you go, m1 and w2, so things are jostling around, just like the round ‑ robin response sends back to our container when it requests to look up the IP addresses for this domain so it can go then reach out to these various containers. And if I open a new tab, paste in the DNS lookup for our tests.echor\_echo, what do you think we're going to get back here when I run this? Let's find out. So we're now on 10.0.3 as our network, and that's because we removed and recreated our stack. And the neat thing is, we get back a list of seven different IP addresses, one for each of our containers. If were to come in here and put in just echor\_echo, what comes back? So, in this case, we just get one IP address. This, again, is going to be our vip, which is a part of our load balancing on our internal echor default network. So just helpful to see how this changes as you scale up and scale down your services. It's a fun way to play with and learn a little bit about the load balancing and even DNS round ‑ robin instead of the vip load balancing.

Scenario: Using Swarm Jobs to Stress Test Swarm Services via Backend Overlay Networks

Now I want to turn our attention to jobs, and for that, we're going to have a bit different setup because we're going to switch back to our raspberry pies. I've taken some time to clean them up a bit so that really all we have right now is the Viz service up and running and I get rid of all the history I could find on anything possible. So we're in a pretty good clean state as if I had just recreated the cluster because what I want to do is go ahead and deploy a set of global or replicated services, and we'll start out small actually, we won't have all five to begin with, but we eventually will. I've got this stress test application, it's a node.js app, it's a cross ‑ platform image, so you're free to run it on VMs or up on play with Docker or on your own setup however you'd like, I'm going to be running it on the pies, and it has a couple of methods, one of which is really fast, doesn't take up much CPU time in terms of the server and the other one has an intentional delay inside of it to slow things down. And as you can imagine, the faster the two web requests is probably going to produce better turn on time and better metrics when we put it under load. Oh and by the way, this stress test web app, I will publish a port so that we can see some of the web requests, but that's not necessary, that's optional. What matters is that port 3000 is what they're running and exposing to the internal networks for the jobs that we launch. Now jobs can be anything. I just thought what a neat example to take a stress test scenario, throw it at a cluster where we launch a bunch of the services or one at a time, maybe two then, and then three, and see what the performance is like, and then why don't we also throughout that cluster some batch work to perform some load testing of those web requests. And in the process, we're going to end up sharing a single custom overlay network. This would be defined inside of our service, so it'll be called sr\_testers, so that's the services side of the equation, and then when we launch the job side of the equation, we'll use the Apache httpd image with its built ‑ in ab load testing command, so that would be our job tier, if you will, and then we'll connect to the same SR\_TESTERS network randomly and perform web requests because we'll be going through that internal load balancer so it doesn't really know which of the services is going to fulfill its request. And then as we get comfortable with one job and probably one service, we'll start scaling each of them up. We'll also add more jobs, for example, and they'll reach out and test other instances of the service. And if our load balancing is pretty even, we should be able to hit all these different nodes somewhat agree to get some interesting test results, hopefully. I hope to see as we scale up, we should see some linear improvements in some of the response times, at least for the one that's CPU bound because if we add more containers to handle the requests, we should be able to somewhat scale up the capacity just like a real life web server. So anyways, this is one idea of what we could do with the job. Sky's the limit. We could be reaching out and testing somebody else's server out on the internet. We could perform some batch work, perhaps some Monte Carlo simulations for financial purposes. Anything is possible. We could perform node maintenance. So we could go into our pie nodes if we gave access to these job containers and let them clean things up, like maybe old containers or images that aren't used or volumes that are dangling. So keep an open mind. What I've come up with for a job scenario is just one of many things you could do. Jobs are just a run to completion concept as opposed to a long ‑ running service concept.

Deploying Our sr Stack with a Service Exposing a Web App to Stress Test with a Job

First up, we're going to deploy our service that we will be testing upon. And for that, I'm sitting inside of the stress folder, it's inside of our stacks examples, and inside of there I've got a stress ‑ replica.yml file. So this is our stack that we're going to deploy, and I've got it all set up so we don't have to worry too much about this. It's going to use this image of mine that I have built with a webstress test suite inside of it. Those two web requests I set up, one slow and one's fast. Internally it listens on port 3000, and we're going to map to port 3000 on the cluster so we have access to it from any of the nodes if we're curious what the requests look like. We're deploying this as a replicated service first, and we're just going to start with one instance to begin with. Now here's what's new. We've listed out this special testers network that I'm clearly and explicitly defining and naming. I could go with the default that comes with this particular service, but I wanted to give it a clever name. I also want to show you you can create your own networks. Now to do that, I need to come down to the networks element that's on par with the services element in a stack file because stack files can define more than just the services. We have implicit networks and we can also have explicit networks. So I've got my testers network, and by the way this will be prefixed by sr, which is the name of the stack that I want to use, and I'm explicitly setting this to be attachable so that I can add on other containers later on and highlight the fact that networks aren't always attachable. For example, the ingress network, you can't attach standalone containers to it. In this case, we're specifying this explicitly, so we will be able to attach just about anything we want to this network, including our jobs. Okay, so let's get this deployed, and I want to call this sr. My context is set to pi1, and we'll run this and see what happens. All right, looks like we've created our network sr\_testers and we've created our service sr\_web. So if I list out networks, you can see we have our sr\_testers network, but we don't have our sr\_default network, and that stands in contrast to viz where we have our viz\_default network. This is the implicit network, instead we have an explicit network. What I'd like to point out in a new tab here, docker network inspect on that viz\_default, look for the word Attachable. You'll see this particular network is not attachable, and that's why I specify that I want ours to be attachable. Now if you inspect our network, so that would be sr\_testers, look for the same Attachable, you can see it's now set to true. So you have complete control over what exactly the network is configured with. You can even set things like the IP configuration or you could add labels if you so choose. And by the way, look, here is a label on here specifying the stack namespace as sr, so it helps the swarm reconcile the current state versus the desired state, should we ever change this network. It also helps to clean up when we get rid of a stack. Okay, so that is our network. And if I do a simple docker stack ps here, I can take a look at our sr stack, and you can see we have one instance up and running as we requested. And by the way, if you're ever curious about services that are available for a stack, I didn't go over this earlier, but there's actually a stack services command, like stack ps, so stack ps is for tasks, well, what if you want to know what services are inside of a stack because a stack could have multiple services, or maybe you just want to know what the names are? Well, you can just put sr on the end here too, and it'll show you this stack's services, in this case our sr\_web service. So you'd have the whole name if you want to access it directly then. And you can see in this case we've got our port published, 3000, so we should be able to make a request to that. Let's open up a tab in the browser here, and I'm going to go to pi1.lan, and 3000 is the port. And then I've also got an inspect endpoint that's a lot like our echo endpoint in our other container. It's just going to print out information about who is actually handling the request. And it's giving us the same information we've seen before, which is local address versus remote address. So here is our VIP, here is the actual container instance that handles the request, and here is the port. So things are a little bit shifted versus the other example, but it's mostly the same information. But that's not really the endpoint we're curious about. Instead, one of the ones I'd like to access is this customer ‑ nodelay and then /1. So this is one of the web requests that has no delay and it just prints this fake customer back. And then the other request that I'd like to make, it's not going to look too different here, it's just /db\_block\_slow, to simulate a slow database connection to return the exact same information. And you're really not going to notice the difference because we're talking about a 10 millisecond delay in there , but it will be enough when we get to testing time to see a difference, especially when we start making hundreds and thousands of requests.

Exploring How to Create Our First Global Job

Okay, so now for our very first test of a job. I've set up this jobs folder, notice not the jobs prep folder, that's just some notes that I have available as I go through things here, I might even remove that before this course is over with, it's a hodgepodge of examples I might want to go through. But anyways, I'm going to copy things over to this jobs folder as we try them out, and the very first one that I want to try out is a 00 ‑ abv.sh. What this is going to do is run the ab command with the version flag, so abv, and I've numbered it 00 so we'll have a sequence of the jobs that we execute, and I've got this inside of a shell script, simply because it's going to be a command that we run and I wanted to make it available for you to copy and paste. I also want to talk about the fact that it might be tempting to try to deploy jobs with a stack file, and unfortunately, that's not possible at this time. That said, there is interest in this feature set where you'll be able to use a stack file as well. There's even a pull request pending with the functionality, tentatively ready to go, and there's an ongoing discussion about this. Jobs were just added in 20.10, which was just late 2020, so, some of the features have yet to land in terms of usability, but the underlying framework is there, and that's a good thing because this is a very long awaited feature. As you can imagine, having long running services is important, and then also having some mechanism to run jobs of various types is equally important. So keep an eye out and see if support lands for using a stack file, and if it does, you should be able to adapt the examples we have here to work with stack files because you already have pretty good familiarity with stack files, we' ve got lots of examples of those. It's just a matter of taking the arguments that we have in the file and putting them in the correct spot inside of a stack file. A couple things about this. I've got a docker service rm up here with abv, which is the name of the job that I'm creating. I just have this up here if you want to rerun this multiple times, you can just remove the job right away, and recreate it. Also highlights the significance of a stack file versus a scripted approach. A stack file both of these pieces of functionality. I consider this the how, and a stack file is more of the what. I'd like this to happen on my cluster, I don't care how it has to happen, if there was a version of the job, well hey, get rid of it. But, we don't have that functionality yet so we need to get down lower into the APIs here and say, hey, remove the job if it exists, otherwise that's not going to cause any problems, it will just cause a quick error and then we'll move on and create the job for the first time, and then the next time you run this, it will do its job and get rid of the old job to create a new one. Now another important thing, as you can see, creating jobs entails using the same set of commands as working with services. So jobs are nested inside of the set of service commands, and at first I was kind of put off by that, but the reality is there's so much in common between the two, it would just be a lot of copy/paste to try and split these apart and have a docker job, for example, and maybe that'll exist someday, but it really won't change how you interact with it. You're still going to do things like name your job, just like naming a service. You'll still need to set the type of job with the mode argument, so, a global job in this case. Or we have replicated jobs, and in this example I'm showing you can still use templated host names, and then next we need to provide the image, and then in this case we can provide a command that we want to run, the ab command, and we'll specify the ‑ V for version. Just a very simple test of running a command that's going to print out more or less just help for the ab command and then stop. The nice starting point for getting a load test up and running.

Running Our First Job and Monitoring Progress

So now, let's go ahead and let's create this job however you'd like to tackle it. I am just going to execute that script that we have. It will error out because there's no job right now, and then it's creating five instances of that job because it's a global job. They're preparing right now, so you're getting nice status information as these jobs are executing. Preparing, by the way, likely means they're pulling the image down. So it's nice to run a global job first because it'll warm up our cluster with the image that we need everywhere. So when we want to do performance testing, we're not introducing the variable of did the node have the image or did it have to pull it down, was it able to partake with all the other nodes in running that same job at the same time to really stress testings, or did we have an asymmetric test where one of our tests ran after the others and maybe a few ran before when perhaps we needed them all to run as close as possible to the same time? This way we know everybody has got the image, which is one barrier to synchronization of the jobs, if that matters. It doesn't always matter, but in the case of testing that can be important. Okay, you can see once things were complete in terms of all of our jobs, and because we asked for a global job, completion will be defined as all five being complete. And notice we also get job complete at the end. And as each one was completing, you could see that we had the word complete show up here. So it went from preparing and then running and then complete, and there might have been some intermediary states as well. Once the job runs to completion, it is not going to run again. And as we're wrapping up this clip, I'm left thinking, you know what, I mentioned that node maintenance is a decent use case for a job. Well, what about just warming up images? Do you need a set of images pulled down to each node? Perhaps that's a great task for a job. It could be as simple as deploy the image that you need, run the help command, but also might be deploy some sort of command that then executes pulling down a series of images, perhaps via Docker Compose or otherwise. So, hey, another great use case.

Viewing Job Output with Docker Service Logs JOBNAME

What can we do to take a look at what exactly happened to see if our job ran successfully to execute the ab command with the version flag, or did we just call something that doesn't exist and get some unexpected error, and we don' t even know it? So, we could use the docker service command to take a look at the logs, and then what do I provide here? Well, we called our job abv, so we just need to provide that right here, and we can get all the logs from the instances of our job that ran. And as I look at the output here, it looks like we've got This is ApacheBench, Version 2.3. So it looks like we are correctly launching the correct image with the correct command to access the A/B testing tool. That's great news. Now, these are going to be interleaved, not certainly in a particular order. For example, here's some of 2, here's some of 5, here's a few of 1, and 3. It looks like these are mostly in order, but you do run the risk of interleaving thing. So another thing you might want to do, that docker service logs command, to get the help for that, you can see we can actually provide a service or an individual task. So how can we get the tasks for our job?

Listing Job Instances (aka Tasks) with Docker Service ps JOBNAME

Again, we are dealing with these service interfaces, so we just need to take a look at the same ps command to get tasks for a job. We need to provide our job, then. Remember what that is? That's abv. Just like that, we can get back our information about the various different tasks that ran or more or less the instances of our job. We can grab the identifiers here, and then come down below, and that's what we can pass to the logs command if we want to look at the logs for a single task. Now one fun thing to do, in addition to looking at one by one logs, we can avoid the interleaving problem if we enumerate through our tasks and one by one call the logs command. And we've got a pre ‑ canned command here to speed this up. This is going to work in Z shell using a compact syntax, but it's more or less a for loop capturing this t variable from the output of this particular command, which is a docker service ps just like we ran before with abv so that we get back our list of tasks for our job. And then I add the ‑ q flag. Do you remember what that does? Well, that just limits the output to the identifier for whatever it is that we are listing out, in this case, our task because we need that identifier to feed, then, to our docker service logs command. And you can see I'm doing that right here by injecting the variable t's value. Now, I added an echo, so you can see what exactly this is going to run before I run it. As you can see, we have five calls to docker logs now, each with a separate instance of a task. So now things will be somewhat ordered. If I take off the echo, I can run this now. And boom, boom, boom, boom, boom, boom, boom, one at a time, we get the log for each task. And now the interleaving problem is no longer a concern. In fact, you can start to see some of the repetition here, This is ApacheBench, This is ApacheBench. We've got the Copyright on both of these, got the Licensed, got the Licensed. Scrolling up here, same pattern. This Copyright Licensed, this Copyright Licensed, this Copyright Licensed. So this would be a lot easier to go through, and I'll shrink this a little bit so it doesn't wrap. This, you can see the pattern now, is a lot easier to understand that we have proper output coming from all of our tasks, and we have no failures on any of them, which is important because it's possible one of them could've failed . So, always feel comfortable using tricks up your sleeve. If you're familiar with a shell language, for example, you want to write a for loop. It's a great way to take advantage of a command that requires a one ‑ off task ID to then quickly use it to exploit it to produce a nice, clean log output. So this would stand in contrast to our docker service logs, passing in abv, which technically does the same thing. It just doesn't guarantee us the same order. For example, you can see down here, we've got some interleaving going on between 5 and 2. And 5, the task is showing which node it ran on. Remember, this is a global job. So one will run on each node. And so while it might be misleading up here that 3 and 4 look like they're in the right order and 1 as well, it looks like 2 and 5 got started almost simultaneously and stepped on each other in terms of output here. Up here, we don't have that problem.

A First Job Instance That Communicates over the User Defined Overlay Network to Load Test Our Service!

So now we're going to run through our first actual load test here of our sr\_web service. So that one instance running now happens to be on pi2. It doesn't really matter. We're going to match that with one job, and it's going to call out and perform a very basic test just to make sure that we can test the functionality, and more importantly, communicate with our sr\_web service. Alright, for the test, I'm sitting here inside of my jobs folder, and I've copied over one of the other examples. We have 01 ‑ ab1.sh. That stands for running the A/B test on one replica. Let's take a look at that script. Just like before, I have the remove, so you can run this over and over. This is named ab1. One big difference is this is now a replicated job, not a global job, because we only want one instance. We don't want five. I specify that explicitly down below, though. That's optional because it's going to be 1 by default. Joining the sr\_testers network again, just like before, httpd image. That's the same. Ab is the same command. Now we have new arguments, though. We have ‑ n for the number of requests that we're going to send, ‑ c for the concurrency. So we're going to send 100 requests with up to 10 at a time, and this is the endpoint that we're going to hit. Notice, we are tacking on sr\_web. This should look very familiar. Let's go into lengths to talk about how our services can talk to each other or, in this case, a job and our service can talk to each other via DNS. This will get translated to what? Well, it's going to get translated to the load balancer that runs on our internal network. It's going to give us an IP, then, that just gets randomly distributed to one of our containers. In this case, we only have one running, so it's going to go to the one on pi2. Of course, port 3000, and then this is the nodelay endpoint. So this should not take very long, and we should be able to see some output when we're done. So, I'll clear out the screen here, and I'm just going to run this. It errors out because we don't have it yet. That's fine. And now we are running our task. It's starting up, and it's iterating through, hopefully hitting that endpoint, and capturing some metrics for us. Okay, notice some things in here. We had job progress, 1 out of 1 complete. In terms of the active tasks, for a while there, there was 1, and now there's none. And now we have 1/1 complete, and our job is marked as complete. You might also note, if you take a look at the service, so I'll list out my services here, you can see we have our ab1. And over here, 1/1 is completed. If you look down below on abv, we've got 5/5 completed. And for good measure, let's make sure that our test actually worked. How can we do that? Well, good old docker service logs and ab1, in this case. We only have one running, so this shouldn't be too bad to look at. And voila, we get our output, and this is what we should be seeing here. This is the timing information coming out of the ab command. So again, this is ApacheBench. It doesn't look like we had any sort of failure. And it's giving some things like the time taken for tests, how many completed, how many failed, how many requests per second. It even gives us a little table of the min, mean, median, and max in terms of timing for connecting, processing, waiting, and the total time. So this is our load test. Down below, probably the most interesting part when we want to start comparing some of these, though, that's really not the point of this course, but you can see we have timing information for percentiles. So the 50th percentile clocked in at 22 milliseconds, whereas 99th percentile, 56 milliseconds. So all in all, very vast in terms of testing time.

Validating Our Job Is Talking to Our Service over the sr\_testers Overlay Network

In this next example, let's assume that for some reason we don't trust that our web requests are actually going through. And let's try to do something to validate that we can send a job out that actually can connect to our sr\_web service and get a response back. For this example, I have an 02 ‑ testy guide. It's really a series of commands. You don't want to fire this off all at once. The first thing it has you do is what I've got done over here on the right side. I've zoomed out a bit, and I've got this command ready to go, docker service logs testy, and then ‑‑ follow. I'm going to create a test job and follow the logs on it. And I'm going to fire off a bunch of commands via it, and I'll show you how that's possible, but we need to fire off one so that we can go ahead and follow things here. If I run this right now, you can see we don't have the service or task yet or job, in our case, so we need to wait for the job to be created. I've already run through some test scenarios, so the image that's going to be used for this is already warmed up across my little pi cluster. Yours will take a bit of time when you first start to run this, and the job is scheduled on various nodes. Don't be alarmed if it takes a little bit of time to pull things down in the preparing state. It will actually give you plenty of time to start up your follow on the logs on the right ‑ hand side here. So, this guide, if I dump this out here, has a series of commands. Here's where you can get the log ‑‑ follow command, and then this is the next one we're going to run. And I'll actually place this over on the right ‑ hand side here in Visual Studio Code so I can copy things out. The very first one we want to run is going to create our testy job. It's going to be a replicated job. There will just be one instance of it. It'll hook up to our sr\_testers network, so simulating everything the same as the jobs we've run thus far. And then it's going to happen to use the same image as the webstress service is using, and that's because this kind of a Swiss Army knife image. I have a bunch of little installed utilities like dig, and curl, and various IP utilities that'll help you troubleshoot or dig into things. So it has multiple purposes beyond just running that web server. It can also run a series of various different command line tools. specifically, what we'll do first is run the dig command and pass along sr\_web as the domain that we're going to look up, much like we passed that into our web app in the last module and got a response back in the browser with IP addresses. Okay, so I will fire this off, and as fast as I can, get over here and start up the logs following, and it looks like I made it. The job over here is starting, and hey, hey, we've got some output here. And would you look at that? We've got 10.0.2.2. That already looks good to me in terms of understanding this is certainly not the ingress network. And I could dig into this 0.2 and figure out what that is. I suppose we could come over here, clear this out, list out our networks, and take a look at what our sr\_testers network has. And it does indeed have our 10.0.2.1, so that is the network for our sr\_testers. So that's validated the first piece of the puzzle are able to communicate via the sr\_testers network like I thought we were. The next piece of the puzzle, why don't we have a little bit of fun here? Why don't we see what it's like to scale up to three instances of our job? What do you think's going to happen over on the right ‑ hand side here? And I'm going to clear this out just to avoid any confusion. So we're going to scale our testy job up to 3 instances. Let's find out. So it's actually starting three separate jobs. When it's complete, the next is started. And if you see here in the output, we've got one unit of feedback, two units, we should have a third here soon, and we've got our third unit. And it looks like all three of our jobs are complete. So when we scaled up to three instances, it took the exact same job that we had and executed it three times.

Scaling Jobs down and Validating We Can Make Jobs That Can Send Successful Web Requests

Okay, continue on. I'm going to clear out both sides here. We're still following on the right ‑ hand side, that's a convenient thing once you get that follow going. Let's go ahead and let's scale down, and what do you think is going to happen when I do this? So we're going from 3 down to 1. Well, what's going to happen is we're going to run one instance of our job. So any time you update your job, you're going to trigger it to be run again. Okay, let's clear this out again, come over to my guide, and what do we have here next? Well, we've got a fun one. This is an update that's going to change the args, which is going to change the command that's run. So instead of the very first command that we had, which was dig sr\_web, this is going to run bash, passing along a subcommand of two commands actually. That's why I needed Bash here. You can see the semicolon in between the two commands. The very first command is a curl call to our sr\_web on port 3000 and then that inspect endpoint because I want to see if we actually get some data back from our website that looks familiar. That's the last piece of the puzzle of testing to make sure we're actually communicating. And then right after that I'm just curious to run an ip as to show my IP addresses. So I'll copy that, come over here, paste this on the left ‑ hand side and hopefully I got all that right, run that, and just wait a second here. And look at that, we've got our output over on the right ‑ hand side. So let's see where does this get broken up at. This is the output of the curl call, and then everything that's not pink is the output of the ip a s. So you can actually see here our IP address is 10.0.2.116. We'll invert this now. These are all of our interfaces down below here. And then up above if we look at the output of our curl call, this is the inspect endpoint, and you can see the person responding is on 10.0.2.3 and the request is going through 10.0.2.107. So you can dig into the network information and see what that means, but at the end of the day the important thing is we're getting data back from one of our web requests. That makes me feel a lot more confident that when we do our load testing, we're actually doing load testing, not just getting that error page back. Let's clear this out again and let's do one more thing. Let's grab this last update download here. This time we're going to execute curl and we're going to hit one of the endpoints that we've been testing, the faster of the two. So I'll come paste that in here and watch for output on the right ‑ hand side. And hey, look at that, it's a bit obtuse with the curl progress indicators, but we have the content here that comes back when we make this request in a web browser. So you can see the same request inside of a web browser externally to the cluster coming in through that published port 3000. And we've got the same little JSON structure, id 1, first name Bob, last name Doe, just like over here at the command line, id 1, first name Bob, last name Doe. So, I'm pretty confident we are hitting the endpoints that we desire in terms of load testing. So let's go ahead and let's clean up here. How do I do that? Well, I can just do a docker service rm testy. Over on the right ‑ hand side you'll need to just Ctrl+C to quit out of that. And we are now good to go feeling confident that we're actually communicating with the service we intended over the network we intended.

Visualizing the Interactions in Our Next Load Test between Swarm Jobs and Swarm Services

Now that we've validated that our tests are accurate, let's go about seeing how things scale. And I've settled on a design where we have one job, and initially we have that one job interacting with just one instance of our service. We're going to look at the slow endpoint on that web application and just see what the performance is like throwing 1000 requests at it, allowing up to 100 at a time. Then, we're going to compare that exact same scenario to having two service containers. We'll still have a job and then we should see something on the order of a 50% improvement or a halving of the time it takes to make the request. And then we will load up five containers for our service and we should see a fivefold decrease in the amount of time it takes to make those requests.

Using Swarm Jobs to Test Swarm Services to Verify a Hypothesis About Scalability

Now, to perform this test, I have another guide file. It's not meant to be executed directly, but it's got a series of steps in it that you can follow along with. It's this 03 ‑ slows ‑ guide. I'm going to close this window because I want to split my screen, and I suggest you do the same if you're following along. On the right ‑ hand side, we're going to do the log following again. And let me go over to VS Code and grab the line here. This is the 03 ‑ slows ‑ guide. I've got a line in here I want you to copy for the job that we're going to use to perform our testing. And specifically, this is a command to follow the logs of our test output on the right ‑ hand side here. Use the same job throughout. It's just going to be one job, and we're looking at the performance of that one job against 1, 2, and then 5 servers, more or less. So, of course, we can't follow this yet. It doesn't exist, so we need to wait. Over on the left ‑ hand side, make sure you scale sr\_web down to one replica. In the event that you've been tinkering with any different sizes, let's make sure that we're starting with the single service instance that we wanted to start with. And I'm going to list out my services. There are quite a few in here. One thing we can do, we can get rid of ab1 and abv. We're done with those jobs. That will help us clean house a little bit here. Now when we list our services, we just have the two, sr\_web and viz\_web. And just confirm that you have 1/1 on sr\_web. And then, let's come back over to our guide. And inside of here, we should have the command to launch our single job. So copy that out, and let's paste it over here, and let's talk through this. So, simple docker service create named onejob. It's a replicated job with only one replica. Just making that explicit here. And then we're going to join up to the same sr\_testers network using the Apache image again, ab. This time 1,000 requests up to 100 at a time, and we're going to be hitting the sr\_web service on port 3000 just like before, but now we're going to be accessing this db\_block\_slow. This one has a 10 millisecond delay inside of it as if we had a rather intensive CPU computation to perform. We should see some sort of improvement as we add more service containers. So I'm going to fire this off and come right over here and follow my logs, and don't worry if you don't follow right away. You'll be able to catch up with whatever has already been printed out. Our job is starting over on our left ‑ hand side, and it's going to perform the benchmarking tests, and you can see that output's already coming through over on the right. It's a little bit small, but we've completed 600 of the requests so far, and we have 1,000 total, so we're just about done here. Okay, hey, there's our output. The 50th percentile's around 1300 milliseconds, and the 100th percentile is 1593 milliseconds. And I'm going to make a prediction at the 50th percentile for our next test. If we double our capacity, we should be able to cut the duration in half or approximately 670. And when we go up to 5, let's just take 1335 over 5. It's about 270. Now, before you run the next test, I' d recommend going to the right ‑ hand side and hitting the Return key a few times to add some blanks between the output of the first test and subsequent tests. If you come over to our guide file, we can scroll down here, and we should be able to update our sr\_web service. And this time, we're going to give it two replicas. I'm also specifying ‑‑ replicas ‑ max ‑ per ‑ node of 1 to make sure that these are spread across separate nodes so that you get the benefit of separate Raspberry Pis, which have separate horsepower behind them. This would be especially important if you have some really computationally ‑ intensive task and you need to make sure that you have the extra CPU capacity. You don't want to have two service instances sitting on one node. So I'll run that, and we will have two instances here in a second. I'll confirm that with the UI, and look at that. We've got sr\_web and sr\_web over here on the right ‑ hand side on two of our nodes, our two worker nodes. I'll minimize this, then. So how would I go about running our test again knowing that I want to perform the exact same tests now, just against two separate servers? We just have to force an update on our job, so we can do a docker service update, put the ‑‑ force flag on, and then onejob is the name of our job. If you want to copy that, that is over in the guide file, the exact same command I just typed in. So now when I run that, our job is going to start testing against two service instances. And you should see output come through on the right ‑ hand side here shortly. Alright, there we go, our benchmarking. And we have 1,000 requests total. That seems significantly faster. Now on the high end, we do have some requests that took longer, but that's the 99th percentile, 98th percentile. That's just probably an aberration. If you look on the 50th percentile, we're talking 646 milliseconds. And not to brag, but if I pull up my predictions, I had 670 milliseconds, so just a little bit faster than that. To be honest, the estimate was mostly luck, but it's roughly an estimate you can make in terms of a ballpark, assumption, of course, you know what your bottlenecks actually are. Let's scale up to five instances of our service. And if you want, that's in the guide, or you can just pull up the command we ran before. Make sure that you have the ‑‑ replicas ‑ max ‑ per ‑ node. Though, I should say, if you set this last time, it will still be there this time. Remember, when we update services, we're only changing the parameters that we pass. So just in case you forgot the first time, then include it this time. Set this to 5, and because we have 5 pies, this will effectively be a global job. Okay, that says that our service is running on all five nodes, and we can confirm that here. We have all five of our nodes having one container. So back here at the command line, now, what do I need to do next as a part of triggering the test? So we want to run the exact same scenario again, which means we just want to force an update again on our job. So do that, and you should see output over on the right ‑ hand side, and cross our fingers that this goes faster, and yeah, that's moving faster than before. And look at that, at the 50th percentile, would you look at that? We've got 256. What was my estimate again? 270. Now this is just a mock scenario I came up with, but I hope you can see how you could easily use this to test out other hypotheses you might have. For example, if you think it's a client ‑ side bottleneck, then you might want to scale the testing side or the job side of the equation and not scale the server side or maybe a mix of both or maybe a network bottleneck. There are different ways you could deploy jobs and services, different combinations of those, increase the number of nodes you have, perhaps dynamically by scaling out into the cloud, and you can learn a lot of things that you probably would have a really hard time learning with just VMs on a single machine where those VMs are limited at the same capacity that the machine has.

Injecting App Config and Secrets at Runtime

Configs and Secrets Are Special Purpose Mounts!

The very last topic I want to touch on really has to do with a few resources that we just haven't addressed that we can add to our stack files that are part of the Swarm functionality. We've seen networks, we've seen services, we've seen volumes even, or even bind mounts to the host. Now, we're going to take a look at configs and secrets. You can think of configuration files as a lot like a volume, in fact, it's pretty much the same concept, it's just a special version of that tailored to the unique aspects of configuring an application and how you need to update those config files. And then you can also think of secrets, for example, key files, as sensitive configuration or volumes that need to be mapped in because they're just files, and so there are special implementation details to keep those secrets safe. Otherwise, configuration, secrets, and even volumes with mounts, it's all really the same thing at the end of the day. And over at the command line, if you look at the docker secret command, this is a management command with a set of sub commands to help you manage the secrets, and likewise, there's a docker config command, and again, it's a management command , and it has sub commands to help you manage configuration files. And in terms of config files, just think of any time you've had an application where you had some sort of XML or JSON configuration that needed to be injected in, well that reasonable to have a separate life cycle versus persistent data, and so it's really nice to have a separate set of functionality around managing configs independent of just persistence with volumes, though you could just deal with them via persistence with volumes just like you could with secrets, but secrets need to be kept safe, so that's why you have these special implementations.

Using an External Secret to Avoid Hardcoding Passwords in Stack Files

First up, I want to talk about secrets. Now I'm sitting inside of the DB folder inside of the repositories stacks examples, and inside of here are two separate stacks. I want to start off by running a diff of these two stacks. Both of these stacks create an instance of a MySQL server. The difference is in how the root password is configured. On the left ‑ hand side, we have a hardcoded password for the MySQL user, and on the right ‑ hand side, we have a new way of defining the password. We're going to use the concept of a secret and to do that, we have to set up a few things inside of our Compose file. So yes, this is a little bit more verbose, but I'm going to show you it's worthwhile. So the very first differences that we have, the MYSQL\_ROOT\_PASSWORD, and then \_FILE, this is an environment variable, we have that pointed at /run/secrets/db\_pass. Now, where does this come from? Well, I mentioned secrets are akin to volumes, basically the same thing. Now in order for this to be mounted here, we have to configure our service somehow, our db service, and tell Docker Swarm that we want that secret to be accessible, also known as granting access to the secret and we do that with the secrets element underneath the db service configuration, under the top ‑ level services element, this takes in an array of secrets, and in our case, we just have one db\_pass. So the name used here is the same name of the file, by default, though, that is customizable. In adding this to our service configuration, we've granted access to our secret. Now we have to talk about how that secret is created. There are two outstanding ways that you can create secrets, one is external to the stack file, the other is you could actually reference a file that has the contents of the secret inside of it. That kind of defeats the point because the file has to be available on the file system then and so we're back to square one with hardcoding our password into a file. Perhaps we can protect it easier than our Compose file, but it's still a file at the end of the day that's not encrypted or protected in any way. So I'm not going to go that approach. Instead, I'm going to set up the secrets top ‑ level element, much like services because these are two separate types of resources. So in adding this secrets top ‑ level element, I'm defining some secrets then that can be used within my stack, specifically by my services. And then I've got d\_pass, the same name, and I've set external to true and that means I need to have this created outside of my stack and I have not done that yet. We're going to get to that in a moment. I have a question for you. What do you think is going to happen if we try to deploy this stack before we create the secret? Let's find out. So we can't create the service until the secret is available. We're getting a secret not found exception. Take note our network was created though, the implicit or default network, but not the service. If I list up my stacks, you won't see any stacks listed, but if I list out my networks, you will see our implicit network is available, it's kind of a weird state. So let's do this. Let's get rid of that stack and that will get rid of our network at least. So now what do you think we need to do? Well we need to create that secret and the best way to do that is with the docker secret top ‑ level command, and this is also a Swarm command. If I run this without args, you will get help. And as you can see, we have a few sub commands, one of which is called create. And of course, if I run this without any arguments, I'll get the help for it, and take note that it requires the name of a secret and then either a file or I need to provide a dash for standard in. So we'll go ahead and create our db\_pass, I'll put a dash here which means standard in, and then I'll come over here and perhaps I'll just echo out some information, something super secret, and there you go . We've created our secret and we get back an ID for the secret. So now we can run the secret ls command. And as you can see, we have a single secret db\_pass, that ID matches the ID we got back here, and if you want, we can inspect our secret and we'll get back basically the same information, but take note, we do not get the value of the password back here. So now I should be able to create my stack. Call that secret for the stack and there you go, we've got our network and our service created. Give this a second and take a look at the tasks involved in our secret stack, and we are still in the preparing state, even though our desired city is running, so that's that desired state reconciliation. Why do you think that is? Most likely, it's because we don't have the image available yet, so it's being pulled down. So let's take a look at this again, and there you go, we are now up and running. So I can switch over to w1, that's the node where our container is running at, and I'm going to exec into the container so we can take a look around. Lazily tab completing and bash, and we're sitting inside the file system.

Docker exec'ing around Our MySQL Container to Learn About Secrets

We're sitting inside of our MySQL container, so we could connect up to our database, no password, and you can see Access denied. And if I put a ‑ p on here, I'll be prompted for a password, pass, and just like that, I have access to the MySQL instance. Could dump out databases, create a database, I'll call this foo, and then I will dump out the databases to see if that's committed, and look at that, it worked. We've got a foo database, so we have root privileges to our MySQL instance via using a secret. Now, let's take a look at how that secret is made available. I'll exit out of the MySQL client, and I'm sitting here in the file system of my container, and what do you think we should look at to find the secret and take a look at its value? Well, I mentioned there's a mount that's provided, so I'm tempted to take a look at the mount command and list out the mounts inside of this container. And I do that, and I look for the word secret. I find one match for a record that links to our run/secrets/db\_pass file. Note this is a tmpfs mount. It happens to be read ‑ only, which means we can't make any changes to it. For perspective, tmpfs is a temporary file storage mount. This is what adds a degree of security, and I should point out this is an implementation detail, something like this could easily change, but it is important to note that on Linux, this is one of the mechanisms to secure your secrets, for example , they're not just copied to your container's file system. But in this case they are mounted or linked in to your file system, so they're conveniently accessible via regular file system calls. And perhaps I should just say in general, the idea with secrets, because they're basically a volume, they are a mount, what makes them special is that they're protected in transit and at rest. So, let's take a look at this file. So first take a look at run/secrets. You can see db\_pass is the only file inside of there. Let's dump out the contents of that, and look at that, we've got our password that I've just set to pass. So, that's how MySQL gets access to our secret, which is our password for our database. Because this is a file, it could be much more than just a password. This could be a certificate, perhaps it's an ssh key. Basically any sensitive configuration of your application can be stored and should be stored inside of a secret. There's one more thing I want to look at inside of this container. And I'll clear the screen here and list out the contents. This is the working directory, when I exe in, you can see there's an entrypoint.sh. Many of the official images have something like this. They intercept arguments that are passed when a container is created and sometimes switch on functionality, maybe run different commands, or in our case, there's actually code inside of this entrypoint, let's open this up to look for \_file in here, there's code in here to adapt several of the prominent environment variables via this file\_env method. This adapts several of those environment variables like the MySQL root user password, so that we can instead use an environment variable with a path to our secret. This is a common and suggested convention that you'll see in a lot of official and unofficial images and probably should consider for your own images. And clear out my screen, and I'm going to leave the container here. Next up, let's talk about removing secrets.

Removing Secrets and an Intro to Updating Secrets with a Link to a Deeper Dive in the First Edition of This Course

So if we run the docker secret command and look at the sub management commands, we have create, inspect, and ls. We have covered all three of those. We haven't talked about removing yet. So a couple things, we still have our secret stack and we still have our secret. What do you think will happen if I try to get rid of that secret before I stop the service that's using it? So we're on docker secret rm, and then we need to provide db\_pass. What do you think will happen when I run this? Well it's going to fail. You can even see that in the output it tells us what's still using the secret db\_pass, the one that we're trying to get rid of. So if we bring down our service first, assume we're done with it, so we nuke our secret stack. Now, what do you think will happen if I try to remove our db\_pass secret? It works. So now if we look at our stacks, we should have none. And if we look at our secrets, none of those as well. It is worth highlighting that you cannot get rid of a secret if it's in use. That said, you can pull a secret from a particular service, so you can remove access to it temporarily, and then get rid of it as long as you add something back or as long as it doesn't need that secret anymore. And when I say add it back, I mean perhaps you're updating the secret to a new value, which by the way, you can't just straight update a secret, that's why there is no docker secret update command. You have to create a new named secret and that's because both secrets and config files are immutable. That doesn't mean you can't change the association for what secret is being used by a service or a config by a service, it just means you need to perform some sort of a rotation, and in the docs for secrets, you' ll find a great section here about rotating secrets, it involves naming your secrets such that they are versioned, perhaps you have an incriminating number on the end, perhaps you put the date on them. And then if you'd like to see an example of this, come up to my first edition of this course, which is quite a bit longer. We tried to trim down this update. My first version was more of a deep dive than a getting started course. That's okay. Lots of people have found great value in it. My first edition has some additional examples that you might want to work through, one of which is a convention for updating a secret where I talk about this aliasing technique. That might be helpful. You might also find other little clips throughout this course that you want to watch one off to complement the latest content we have here in the refreshing course. If you ever have trouble finding my first edition or anybody's first edition to any course, the author page has all the additions of a course with direct links so you don't have to search for them. I also put links into the course repository where the sample code is at so you can get to both.

Secrets and Configs Are Parallel Concepts - Secrets Are Merely Sensitive Configs

When the time comes where you want to use secrets, that's the time to read through a lot of this documentation. So the last thing I want to do is point out some good resources here for secrets, and then inside of here you'll find links. So if you get to any one of these docs pages I'm referring to, you're going to find links to the other docs pages that I'm referring to. If you search in here for compose, you'll see the Compose file reference for details. If we open that up in a new tab, this is Compose file version 3 reference, and you can see we were brought to the anchor for secrets. This is granting access on a per ‑ service basis. So that's where we say, Hey, this service has access to this secret. There's also then a link here to take you to the top ‑ level secrets configuration. That's where we literally are at the top level defining the secrets that we might then grant access to. And just like this exists for secrets, if you look in the same reference putting config, you'll find a configs anchor. And it too has a link to, within the same document, the top ‑ level configs section. So of just repeating basically the same demonstration instead for you, I'm going to leave it as a challenge to you to explore config files. They're the non ‑ sensitive equivalent of a secret. See if you can't do what we did with the secret where we got into the container, we looked at mounts, we found where our secret or in this case our config is mounted at, and then we just played around with those management commands for secrets, well, play around with the management commands for configs. And then there's probably one other resource for configs that I want to point out. It's the guide like we had with secrets. I think it's linked off of this. Yeah, it is. This is just a sub ‑ portion of the guide. There's a store configuration data using Docker configs just like there was one of them for managing sensitive data with Docker secrets. These are parallel docs. They'll probably read a lot alike, but they go over the nuances of each mount type, if you will. I'd say first try to figure some of this out on your own, just thinking about what we did with secrets, and see if you can map it onto configs. And then come here and read through this if you can't find something because there are subtle differences between the two in terms of, for example, where things are mounted at. And with that said, this is the end of the course. I have really enjoyed presenting on this topic because Docker swarm is one of my favorite tools out there. It's just so incredibly easy to use to cluster machines together. As you can see throughout this course, we didn't have to perform a myriad of configurations or generating certificates or all that insanity. We could just get up and running. There's even Play with Doctor if you don't want to have to deal with even building out a simple cluster in the few commands you have to perform to do that. Maybe even try to use it in a development environment or even a CI environment as a next step. There's definitely value to be reaped out of it because you can expand beyond the capacity of a single machine in both of those contexts and actually expand up on a really large level, for example, in a CI environment. And then from there, if you like what you see, just take it through to production. If you would like to reach out to me, I've got a form on my website. Looks like I need to update it to be not so small, but you can put your information in here, and I'll get back to you via email. And don't forget, the course page exists. Go out and have discussions, have discussions with me, without me, help other people learning, and ask your questions and see if anybody can help you. And I hope you enjoyed this course, and I'll see you next time.