Course Overview

Hi, my name is Wes Higbee. Welcome to my course Implementing a Self-hosted Docker Registry. From a high level, containers and images can seem mystical, thus all the components of Docker, for example, a registry, can seem overwhelming to understand. However, I've found that once I get under the hood and take a look at how things work behind the scenes, the complexity melts away. And one of the best ways to look under the hood is to run your own registry. And of course, self hosting affords a lot of benefits in having total control over a registry. We'll see these benefits by first deploying our own private self hosted registry and using it to push and pull images. We'll see up a registry mirror of Docker Hub. We'll look at Webhooks to trigger subsequent activities when new images are pushed to our registry, and then we'll get into some of the nuances of running your own registry. For example, cleaning up images and choosing a storage backend. We'll also talk about securing a self-hosted registry. And finally, we'll look at deploying a registry as a part of a production cluster of applications. Let's get started.

Deploying Your First Registry to Distribute Images

The Journey to Self Hosting

When I say Docker, what comes to mind? Chances are containers come to mind, but containers are only a small piece of the puzzle of what makes Docker great. I really don't even thing they're that interesting when compared to images. Image distribution is what led to the explosive growth in interest both in Docker and containerization in general. Containers have been around for a really long time, they just weren't very easy to use. And even though we had tools like LXC that predate Docker, they didn't take off because, for the most part, you're still stuck creating your own images. So as a mere act of uploading to a server, a series of common images that people could benefit from, making those available on the cloud via Docker Hub so that anybody in the world can reach out and grab images, all that's necessary is to install the Docker CLI and execute a Docker run along with the image name. And within seconds you can go from knowing nothing about Docker to benefiting from it. Because with Docker, it's now possible to pull software down to your personal workstation and run that software without needing to know how to install it, how to configure it. Instead, you just need to know what software you want to run and then you can run it and use it and it's not just your workstation, obviously this applies to servers. And of course, the implications of this for software distribution are radical. We're really at a point where you can install one piece of software, Docker, and then it will take care of installing any other server software, even desktop type CLI software that you want to run, and maybe other types in the future. Of course, the distribution mechanism itself is technically important, but it's also important that you have an ecosystem of people that are contributing images. If you spent any time at all using Docker, it doesn't matter what you're using it for, chances are you've realized that you can do a lot with Docker Hub. But at some point you might stumble on the fact that there are other places where you can find images, for example, Quay. io, and there's even this notion of Docker Store, not really separate, and that provides a commercial set of certified images on top of the community images that we have in Docker Hub. You can even push and pull your own images to many of these services and never need to host your own registry. That's really impressive because a lot of the times that's free to do, especially if it's a public repository or a public set of images that you're willing to share with the world. To think about the storage and bandwidth implications, it's pretty impressive that these things are free for us. At some point though, you might have images that are sensitive or, for a myriad of reasons, you might want more control, in which case you can set up your own registry, perhaps internally or a server that you spin up in the cloud, and you can push and pull images form this registry. And it's common to refer to this as a self-hosted registry. And of course, we're going to be talking about self-hosted registries in this course. As a very first step I want to show you how to set one up, for two reasons. Number one, I want you to see how easy it is and number two, I want you to get a feel for it, and then with those two perspectives, we'll come back and talk about use cases, what all you can accomplish by self hosting. Now before I switch over and do a demonstration here and walk you through things, I've got a question. In a minute here when we run our own self-hosted registry, how do you think we're going to do that?

Running the First Registry

Sometimes I feel it's rhetorical to ask how we're going to run some piece of software with Docker. At the same time, I think it's important to point out and kind of fun to reflect on the fact that containers and images and Docker and this entire ecosystem are rather fractal in nature. And what I mean by that, well there's an image out on Docker Hub, a registry image, and this is an official image, when you pull this image down to your computer, you can use it to start up your own self-hosted registry. And of course, you would do that inside of a container. That's a beautiful thing and that's why it's so easy to do. So let's go do this. And if you really would like a challenge, knowing the name of that repository is registry, go ahead and try to start it up on your own without my help. Here is the official registry repository and at the time of recording, the latest version is 2. 6. 2. Now this is a registry that we need to interact with, so we'll need to communicate somehow, which means there's probably a port that we need to be concerned with, and of course, if you scroll down you'll find in here the port number is 5000 inside of the container, that's what the application listens on. And we'll go ahead and do the same thing here, we'll publish it to the host on port 5000. So here's my call to docker run. I'll get rid of the container when I'm done and I want to see the output of the registry process it runs, so I won't run it in the background with -d, instead I'll run interactive and, of course, publish that port we just discussed, 5000 in the container to 5000 on the host, and this is the registry image. If you want, you can be specific about what version to use, as long as you're using something in the 2x series, that should be fine. 2x has been around for a long time now, there is an older v1, stay away from that. Okay so let's spin this up, pull down the image, of course if you don't have it, it's pretty fast for me. And of course, let's just ignore all this output until something goes wrong. What could I do to interact with this registry now? Well it's running on port 5000, so let's take a look at that. If you hop over to a browser and load up localhost port 5000, assuming you're using Docker for Mac or Windows and you've published that port to your host machine, otherwise pull up the server that you're running the Docker daemon on, go to port 5000, you'll be disappointed. If you look into the request itself in the developer tools, you'll see that there is no content-length. We're just getting a 200 OK. If, however, you come up here and put in slash and then v2 slash, and then \_catalog, now you'll get a response back and you'll get a JSON payload. I happen to have an extension in Chrome called JSON viewer that is formatting this nicely. You're JSON might not look the same. And in the response you can see that we have an empty array of repositories here, so we have an empty registry at this point. Now what could I do so that I could see something in this list of repositories?

Registry Organization Terms and API

To answer that question I want to take a quick detour and talk about a few terms involved with the organization of how images are stored inside of registries. So we have this concept of a registry. Examples include Docker Store or Docker Hub and also that self hosted registry that we just spun up. Inside of each of these registries are a series of repositories. For example, out on Docker Store, there is a SQL Server repository for Linux builds of SQL Server. There's a repository for NGINX, and there's also a repository for BusyBox. So various tools that we might like to use. We can pull down images from these repos then because each repository is compromised of a collection of images. And it's pretty common, although there's no rule, it's pretty common that a given repository will house a series of related images. So in this case, we'd probably only find images with SQL Server, not going to find NGINX in the SQL Server repository. If we want NGINX, we'll need to look inside of the NGINX repository. So inside of there, we'll find another collection of images, and so on. And then to help us mere mortals find the image that we want, we use tags to locate a given image inside of a repository. For example, at the time of recording, in the SQL Server repository, there are a couple of images, one tagged with 2017-latest and one tagged with 2017-CU1. There are no rules about what tags you use. Each repository can use its tag namespace as it sees fit. It's pretty common, as you see here with the SQL Server repo, and also the NGINX repo, here are some of the tags, it's pretty common to use the version of software to tag different images. So if, for example, I want NGINX 1. 10, I can specify that as the tag. Or if I want the latest version of NGINX, I can use 1. 14. 0 at the time of recording. Tags are references to images, so it's also possible for the same image to have multiple tags that point at it. And using the name latest is a pretty common convention to pull down the latest version of a piece of software or often the latest stable version of that software. Of course, again, the tag names are entirely up to the people that are responsible for a repository, so latest may not be the latest version. Keep these terms in mind throughout the course. And one trick that's helped me keep all of these organized in my head, especially because registry and repository both start with the same two letters, if you look left to right here, you can see a series of one to many relationships. One registry can have as many repositories, one repository can have as many images, and a given image can one or more tags that point at it. And then zooming back out to the left most side here with the registry, one last note is that what defines a registry is the API that we interact with to both query images and push and pull images. The registry API is an HTTP API and we're working with version 2 of this API. And if you're curious and would like to know more about the documented behavior or specification for this API, check out this HTTP API V2 set of documentation. As you can see, it is rather large, an estimated 126 minutes or about 2 hours. You don't need to read through this, but you can refer to this as you have questions. If you do want to read through some of this, I would recommend the introduction and the overview and then stop when you get to the detail section and refer to the detail section when you have specific questions about the API. Like if you want to know more about an error code that comes back. Now that you understand this organization scheme, and you can see our catalog returns back to us an array of repositories, then the answer to that question I posed before, how can we get something to show up here, is to push an image. Let's do that next.

Pushing an Image to a Registry

So I can come over to the command line and I can do a Docker push then to take an existing image and put it onto my registry. So docker image ls here and I'll be specific, I want to look for mongo images. I'll take this mongo image and I will push it out to this new registry. Can you take a guess at how I can push this mongo image to my registry? Not to Docker Hub. Well I won't be able to push it to Docker Hub, but what could I do here to say hey Docker, would you please push this out to this new registry that I just set up on localhost port 5000. I won't fault you if you haven't worked with a registry besides Docker Hub, this isn't necessarily intuitive, but what you do is just tag your image and give it a new name and that name needs to contain the registry that it's going to belong inside of. So in this case, I will take the mongo image, localhost port 5000, slash, so you put the registry first, and I will go ahead and just keep the mongo repository, and if I want I can specify a tag. I won't do that in this case, which means that the image will default to latest. By the way, take a moment to write down this format for naming your images and refer back to this throughout the course. As we push and pull various images to our private registries, understanding these components of the name will help you keep things straight. And I should emphasize, like the tag, the registry is optional, though in our case whenever pushing to a private registry, we will need to specify that. Also, the registry port is optional. As we will see later on, depending on if we're working with a secure or insecure registry, over HTTP or HTTPS, will dictate if the port is by default 80 or 443. Now to help solidify this naming structure, let's go back to our command here, and now that we've broken down the new image name, let's break down the source image name. And the very first part of breaking this source image name down is the simple question, which component does mongo map to? Is it the registry, the port, the repository, or the tag? Well in this case, and in every case, a repository is absolutely necessary. It's the only not optional component. So mongo maps to the repository that we're pulling an image from. A couple more questions then, what registry is this mongo image coming from? So if we don't specify a registry, then the registry is defaulting to docker. io. Essentially at the end of the day, we are pulling from the public Docker registry if we don't specify a registry. And the last part, what tag are we referring to? Well if we don't specify a tag, then the tag is simply latest. Now you might we wondering, how do you figure out what these defaults are? Well you can definitely refer to the documentation for Docker. I also find it extraordinarily helpful just to peruse the source code, the distribution GitHub project out under the Docker. org/distribution. Peruse the source code here for much of the behavior of interacting with registries. For example, you can see here, a cursory search for defaultDomain points at docker. io. We can also see the defaultTag here of latest, and if you peruse the rest of this file, you'll better understand how image specifiers are processed. Open source is a beautiful thing when it comes to learning. So whatever image latest points at in the mongo repository, in the public Docker registry, that is the image that we will pull. I find it helpful to keep these implicit values, or default values, in mind when breaking down image names. Okay so let's go back to our tag command. What do I do next? Next is just a docker push. You can see the image is being uploaded now to my self-hosted registry. Now that it's pushed out, what can I do to confirm that it pushed successfully to the correct registry? Come over to the browser and just refresh here. You can see we now have a repository.

Configuring the Docker Daemon to Allow an Insecure Registry

In addition to pushing images, we can, of course, pull images off of our own self-hosted registry, and to do that I have hopped over to another computer here and I'll quick list the images and look for mongo images, and you can see there are none. And for fun, I'm going to run a ping here of both Google. com and then the node inside of my network that has the registry running. I'm doing this so you can see that when I pause internet access for this Windows machine, that we'll still be able to pull images from our local registry. So I'm going to go pause now and you can see, there we go, we've got our request timed out for Google. com. So you can see on the right hand side, pings are still going through, but on the left hand side they're not. So I've shut off internet access, I can't possibly be pulling this image from Docker Hub where the mongo image also exists. And of course, this might happen if you lose your internet connection. But, because we have our own registry, which I can get to with a simple curl here, so I'll find my node IP address, and you'll have to do the same. Go to port 5000 and v2 and then /\_catalog. There you go. You can see we get back our list of repositories. So I have access to my self-hosted registry. You might have to check firewalls if you have any trouble doing this, and so next up, I do a docker image pull, and then what do you think I specify here? Well before it was localhost, that's not the case anymore, so instead I need to put that IP address of my registry, port 5000, and then just /mongo, like before. Obviously in real life you probably want a domain name associated with your registry for consistent access, both on the registry itself, as well as remotely, but also so people don't have to remember IP addresses. Now when I run this pull here, you can see we have a problem. The server gave us the HTTP response, the client is expecting HTTPS. And that's because the registry we started up is an insecure registry right now. We have not yet configured security on this registry. And if you run a docker info, and if you look down below you'll see an Insecure Registries section. Likely you only have loopback or your localhost 127. 0. 0. 0 here, I also have another registry I've used in the past, so that's configured. But because our registry is not in this list, then the daemon is going to refuse to pull images from it. So we have to configure access to an insecure registry on another machine. Now this configuration is for the Docker daemon itself, which pulls the images and pushes the images, not the Docker CLI. And how you have installed Docker will dictate how you can configure these registries. So if you decide to go the route of just allowing an insecure registry, perhaps because you have restricted access to that registry, then you'll want to configure the daemon, and in my case here I'm on Docker for Windows. Do you want to take a guess at how I go about adding an insecure registry? Well in the Docker for Mac and Windows, if you come into the settings for Docker and got to the Daemon tab, there's a list of registries that you can add if you're in the basic configuration mode. So I can just come right in here and paste in, there we go, there's the IP: and then the port number. If you go the route of the advanced config in the settings here, or you're using a config file on disk, you would just add this to the insecure registries array. Anyways I'll flip back to Basic here and then make sure you come down to the lower right corner and click this Apply button, that will require a restart. And then once Docker is back up and running, confirm that setting with the docker info call, and there you go, we've got our registry added as an insecure registry. So if I clear out the screen now and run my pull again, now you can see that the layers of the image come down to my machine. And keep in mind, I'm still blocking my internet access right now, and I'm able to pull these images, and that's because my registry happens to be local. So this hints at another possible benefit of self hosting a registry. Alright, got that pulled down now, if I look at my image list now, you can see we have that mongo image.

Distributing Images Without Access to Docker Hub

So now that we've pushed and pulled one image, let's try this again to test your memory of what all we did and start to solidify these new concepts. So over on my Mac where I have the registry running. You can see I have the NGINX image, just like I had that mongo image, pulled down. So if I want to push this NGINX image to my registry and add it to the list next to mongo, what do I do first? Well first up, I need to tag the image, give it a new name more or less, so the source is nginx and then localhost:5000 and then I'll also call this nginx, maybe I'll add a 2 on the end here just for fun and once I've done that, I can look through my images for NGINX images, actually I have three here, some other testnginx, you can ignore that, but right down here at the bottom is our new image name or tag. Now with that, what do I do next? Now we do our docker push, and instead of mongo, nginx2. So that pushes out to our registry. Over on the Windows machine, what can I run here to just check the repositories out on the registry, so the private registry? Well that's where the curl command comes into play. Now before I run this, a few side benefits I'd like to point out. First up, I'm using the reverse search in PowerShell, which is also available in bash for Mac and Windows as well. You can activate it with Ctrl+R and as you type, it will search backwards through your command history. And then you might be wondering, what is this curl command? Isn't that a Linux thing? And it is, but PowerShell has an alias to make this operate, behind the scenes it uses invoke web request. The nice thing is with simple calls to curl, you can just ignore all of that and use curl if you're familiar with it. And of course, this works across Mac and Linux as well. Or you can natively access the curl command. Alright let's run this, and there you go, we've got two repositories now, nginx2 and mongo. So what's next in the order of getting this NGINX image onto my Windows machine? Well I can run a pull again and this time just put in nginx2 on the end. And I could even run a container based on this. And then since this is NGINX, I'll need to publish a port, I'll do 9005 into 80 inside of the container. I'll clear out the screen here and we'll run this container. And when I pop open a browser and go to that port on 9005, I have my NGINX server up and running. And, again, right now I don't have internet access on this machine, so I'm using just the registry to get my software into execute it. To wrap up this demo, Ctrl+C will kill off NGINX.

Running a Registry Web UI

By now I'm sure you've noticed that we have a nice API to access our registry, this API allows us to push and pull images, as we've seen from the command line. And then we can come check the registry via this API to see what it contains. For example, here's our list of repositories. I could also open up a new tab and I could navigate to one of the repositories, the mongo repository, and list the tags for that repo, and you can see we get back the latest tag, which is the one that we pushed. And we could do the same thing here, if I come over to the catalog, we could copy nginx2, come over here, paste that in, and I want to leave the mongo tags open down below, so I'll use a handy shortcut. This is a Chrome shortcut, and it is Command+Enter on a Mac or Alt+Enter on a PC. It will take whatever you've typed into the address bar on the current tab and it will open it in a new background tab, and then it'll change your URL back to the point before you made the changes. So you can here now in this tab, I'm still looking at the mongo tags. And over in the other tab we have the nginx2 tags. So it's this registry API that we're working with that forms the centerpiece of a registry. However, on a daily basis, you probably won't work much with the registry API directly. Chances are, you'll push and pull images from the command line or if you do need to search through the images that are available in a registry, it might be kind of nice to have something like Docker Hub where you can go to a website and type in search terms. For example, with Docker Hub, we can come out here and type in NGINX and we get back a nice list of repositories to look at and some information about those, and we can drill in and find out more. You can have the same thing with your own private registry. It's important to understand though that this is not a part of the private registry. So the image that we pulled to run the registry, that's just a standalone registry with an API to access it. So knowing that there's no web UI bundled, what do you think we might do instead? Well we have this API sitting out here, it's possible that we could have some sort of registry server running, a web UI, and if you search for a registry UI out on Docker Hub, you'll find a lot of possible images to use. If you scroll down a bit here, I want to use this docker-registry-frontend repository. This happens to be a front end that I like. You could use a different front end, nothing wrong with that, and one thing I will say is whatever you pick, just make sure that it's safe to be using and that it provides the functionality that you would like. So this is a separate image that we can take and spin up a second container and then connect it back to our registry API and it will allow us to browse images, somewhat like Docker Hub. And of course, if you scroll down, there are some nice usage instructions in here. The gist is, we need to point this registry front end back to our registry API, both the host and the port. So let's do a docker run here. We'll make this a background service. It might be helpful to name this. Maybe we'll call this registry-ui, and I'll wrap a line here, and then first up I need to specify the host name for the registry, and in my case I'll use the IP address of my local computer here and I'll also specify the port of 5000. Of course, the front end here will host its own website, so we need to publish a port for that, so how about we publish 8080 on the host into 80 inside of the registry front end container. And the last, we need to specify the image. So that's the image we found out on Docker Hub. And in this case I'm using the v2 tag. So the key here, make sure you point this at the correct host name and port for your particular set up. When you run that, then we can hop over to the browser and if you pull up the localhost port 8080 or whatever you port you publish to, you'll see, we have our Docker Registry Frontend up and running. And this particular tool allows you to browse your repositories. You can search through them, you can click on one of your repositories, drill in and see the tags, click on the latest tag and drill in and see some information about this particular image. You can get this nice bread crumb organization to navigate around and this is obviously much more helpful than just poking around an API and hoping you remember how to format the API request to get the information you need. So if you're deploying your own private registry, keep this in mind. If you want people to browse images, then this will be a helpful addition to your toolkit. It's not necessary, again, because for the most part when it comes to a private registry, it's pretty common just to be pushing and pulling images. Even with Docker Hub, if you think about how often you go out to Docker Hub and do things versus just pushing and pulling images, you'll probably find that most of your time is spent at the command line. So this is completely optional, but it can be valuable.

Beware of the Default Anonymous Registry Data Volume

We've now got a pretty clear picture of how to start up a registry and maybe a front end for that registry, we've covered the basics. So now let's step back and do a little bit of housekeeping here. And the first bit of housekeeping that I'd like to cover is how we go about starting the registry. So when I start this the very first time, I added the remove flag. Nothing wrong with that when you're learning, but that's going to clean up all of the resources associated with this container, including a volume right now that is storing the images that this registry holds. That's going to happen the second this container stops, so if you restart your computer or lose power, for example, all of the sudden you've lost your registry and all of your images, and let me show you that. So right now, here is my catalog and you can see we have our two repositories. If I hop over and kill off the registry, a simple Ctrl+C will do that, now back over in the browser, when I refresh, obviously we get nothing and then if I start that registry container back up, we have an empty list of repositories, so they're all gone. So it's important to know, if you hop out to the Docker file for this registry image, it's important to know that there is a volume, by default, where your images are stored, and this is an anonymous volume, by default. It's defined here in the Docker file. So if you do nothing to protect this volume, once it's removed you lose all of your images. That may not be the end of the world, in fact that can be a really nice thing when you're learning and you want to just wipe out the environment. It might even be nice in an environment where you don't want to keep images long term, you just want some sort of transient registry, maybe in a continuous integration environment where you don't want to keep state around very long. But if it's desirable to keep your images, which chances are, if you're setting up a private registry of some sort that's going to hold production images, well then storing those images and not losing them is probably a pretty important thing.

Observing the Default Anonymous Volume Lifecycle

So what could we set up here to protect this volume so that we could destroy and recreate the standalone container if we want to and not lose our registry of images? Well we could create a named volume instead of an anonymous volume. Also, later in this course we will talk about other storage options besides storing images to disk. But for now, let's go ahead and set up a named volume. And if you'd like a challenge, why don't you try to set that up on your own and then come back and I'll walk through things. Okay so over at the command line, let's kill off the registry. And I want to take a moment here and just take a look at the containers that I have. By the way, I've changed the format of my ps output to not wrap so much. I dropped some of the columns. Let me show you that quick. So my docker config file, which controls settings for the CLI. Inside of here you can see I set the psFormat and provided a format string, just like you would with the CLI directly. So now this will be my default. I didn't want you to be thrown off by that. Alright, I also want to list my volumes. I have two volumes. I happen to know that these are associated with that front end container up above, but let me show you how I know that. I've got a helper alias here, I'll expand this so you can see what exactly I'm passing here, just a special format string, this time very similar to my new format string except I'll add Mounts and drop the status column. I'm doing that because you'll now be able to see over on the right hand side that there are two volumes associated with the registry front end container. Both of the two volumes that we have here, and both of these are, well, you tell me, what type of volume are these? Are they named or anonymous? So both of these happen to be anonymous and that's why we have the random names assigned to them. I wanted to show this to you because now I want to start up that registry container again, so do a docker run here. I'll leave the same kind we have before with rm, which will clean up anonymous volumes once the container is stopped. So we'll just launch this and I'll split the screen here so that I can run the volume list command again. Now you can see we have three volumes and I'll also list out the container to volume relationships. You can now see that the new anonymous volume with 77 here, that is also above, is associated with our registry container. And of course, if I stop the registry container using its container ID 135, the start of it, what do you think is going to show when I list the volumes? Is that what you expected? So you can see the 77 is gone, it's an anonymous volume, it's just wiped out once we stop that container because I passed the rm flag up above here, which destroys anonymous volumes.

Storing Registry Data on a Named Volume

Now that we've observed the life span of the default anonymous volume, let's take a look at using a named volume. And just clean this up a little bit here. And I'll list out both of our containers and volumes again so you can see where we're starting at. Now when I make my docker run call, if I modify this to add one additional volume flag, I can now specify a named registry called just registry, or whatever you would like. I could put a -data on here, just to avoid any confusion that this is not referring to the image. Okay, so I've got my name there for my named volume and then a colon and the path inside of the container where this will be mounted at. Quick quiz, where can I go to find what this internal path needs to be? Well we saw this a moment ago inside of the Docker file. We can use this exact same path here. It's a default location that works with the default configuration for this particular registry to store images to disk, so I'll specify that instead, and now when I run this registry, a couple of things. I can split the screen and I'll scroll up here so we can see the volume history up above. What do you think we'll see when we list out volumes now? And there's our new named volume. And I'll go ahead and list out the containers here and take a guess at what you think you'll see in this output. What are the associations? Is that what you expected? So as you can see, our registry-data named volume is associated with the registry container, that makes sense, that's what we passed at the command line to docker run. And now the million dollar question is what happens if I stop that registry container now? So here's the ID, what's going to happen to the volumes now? Let's stop it and find out. Drum roll, you can see we still have our volume sticking around. You will have to explicitly remove this named volume with docker volume rm. And so the neat thing, if I hop up above, maybe make that a little bit bigger, I can run the registry again. I'll come down below and I will push the NGINX image back out to our local registry here. And if I come out to browse with our frontend, you can see we've got nginx2. And for good measure, let's push another image, how about the mongo image. And let's flip back to the browser and refresh, and there you go, mongo shows up. So we have two images out here, now let's go about the process of destroying and recreating our container and make sure we keep these images in our registry. So back at the command line, and then I can go ahead and stop the registry container. As you can see, we just have our front end running now, but what about our volumes? We still have our data volume. Now let's clear this out and recreate a registry container based on the registry-data named volume, and let's see if we have our images. Now that that's started up, let's go to Chrome and query the API, and there you go, we have our two repositories with our two separate images. And out on the browse tool, I can now access the images here. So you're now prepared to defend the images in your registry should you care to. I would like to point out that this storage and this particular directory that we mounted a volume into, this whole strategy is based on the default configuration of the registry image, so there's a config file for the registry, it's based on this config-example. You can see this is copied in from the root of this repository, so I can actually come up here, go down below and check out the registry folder and look at this config-example. yml file and inside of here, you'll see there's a storage section, and we'll talk more about this later on, I just wanted to show this briefly. And inside of the storage section, you can see there's a filesystem configuration section. The rootdirectory is specified in that same location that we mounted our named volume, the same location that was in the Docker file too for the anonymous volume, by default. So because of all of this, you're good to go, as long as you stick with a named volume or just be very careful with how you manage the volume associated with your registry.

Why Self Host?

As we wind down this module, I want to take a moment and touch on some of the reasons why, some of the considerations behind self hosting a registry. In this first module, we've already seen how we can easily spin up a self hosted registry and use it to distribute images. We even saw how this can facilitate offline image access. In the next module, we'll talk about using a registry mirror to cache images from Docker Hub and make them more readily available locally. This can provide an offline cache, but it can also boost access to images for the first time, especially if you have a lot of clients pulling in images, having a registry mirror may mean that you only need to pull an image one time to fulfill the needs of many of the clients on your network. That's going to save on both bandwidth and time spent waiting to pull an image. Later on in the course, we'll take a look at how we can set up notifications to trigger actions when images are pushed to a custom registry. For example, what if you had a security team that vets images from Docker Hub so that you're not running any old image willy nilly inside of your private network? So the security team pulls down images, vets them, and then pushes them to a security vetted registry hosted inside of your company. And then perhaps after pushing the vetted image, you have some images that you want to build on top of those, that's where notifications can come in and kick off additional processes perhaps to build other images that are pushed out to other self-hosted registries. So it's entirely possible your organization may have multiple self hosted registries that belong to different teams of people that use them for different purposes. You could have distinct islands of responsibility where you delegate control to individual teams. As we drill into notifications later on, we'll also see how we can trigger a job inside of Jenkins, which brings up another potential benefit. If you're establishing a continuous integration and/or deployment pipeline, and your artifact for deploying your application is an image, then a custom registry is a great place to house the various builds of your application as your application progresses through your deployment pipeline. You can have that registry sitting right next to your CI server. Security is another consideration. Running a self-hosted registry gives you both flexibility and total control over how security is implemented. And one last consideration, self hosting a registry is a great way to build a fine tuned cluster in your production environment that can support your operations as you scale out to many, many more servers. We'll talk about this toward the end of the course. With these benefits in mind, let's get started. Let's take a look at how to do all of this and help you find ways to take advantage of these benefits in your own work.

Registry Mirroring with a Pull-through Cache

The Value of a Local Registry Mirror

Bandwidth is one of those things that you can just never seem to get enough of, especially when you're working with Docker images, which can easily get into the hundreds of megabytes, if not gigabytes. And unless you have a Verizon Fios gigabit fiber symmetric connection with 940 up and 880 down, and that's megabits per second, not kilobits, unless you have this, chances are you're going to experience some latency when you're pulling Docker images. How do I know this? Well even though I seem to get fliers every other week in the mail from Verizon Fios for gigabit internet, even though I get those fliers, every single time I call them and I say, Hey will you please take my money, sadly they reply that they are not providing service to my building. They provide service in the building next door. Mental note, next time I move I should ask about ISPs that are available in my building. So unless you're blessed with some big pipes, pay attention to this module. There's a really neat feature of Docker registries that will help you out. And that feature is called registry mirroring. At a high level, normally we pull images from Docker Hub and we're subject to whatever speed of connection we have available. The closer we are in terms of latency and the bigger the pipe that we have, the faster we're going to pull that image down. And of course, the further away we are, and the slower the connection we have, the longer it takes to pull down our images. So instead of pulling directly from Docker Hub, we can set up another registry and then we point our local Docker daemon at the registry mirror as if it were Docker Hub, so that we send image requests to our mirror. The mirror will look in its cache to see if it has that image already and if not it's going to request the image from Docker Hub and that might take a little bit of time. Once the image is obtained it's cached locally, and of course sent down to our daemon. That way the next time we go to make a request for that same image, because it's already inside of our cache, our registry mirror can respond right away and send us the image. And because this mirror is local, it will be a lot faster than reaching out over the internet, even with the fastest of connections, but especially so when you have a really small and long pipe. And it's not just one device that can take advantage of this, this registry mirror can be available to all the workstations on a local network. That might be a home network where you have multiple laptops and desktops, each of which can make a request for images, multiple at a time perhaps. And if it's the same image that's been pulled before to any of the computers on your network that use the mirror, well then the access times will be wonderful. Even if you lose your home internet connection, you'll be able to pull images that you already have, which is actually where I stumbled upon this use case. Back in my early days of teaching people about Docker, I'd have a classroom full of 20 or 30 people and inevitably the conference that I was speaking at did not have the best of internet. Even if they had broadband, maybe at 10 or 20 megabits per second, it just is not enough to be able to get people to pull images fast enough to learn much and try much inside of a 4 or 8 hour workshop. And so I would bring with a registry mirror that had all the images already downloaded on it, and it would be a fallback, as long as the internet was fast enough we wouldn't use it. It was a nice way to be able to guarantee that we could do some really neat demonstrations even with large images. But it's not just workstations, of course this can benefit your servers. In fact, you'll probably collocate a registry with servers closer to the servers than your workstations, or maybe even a dedicated registry for your servers, which, of course, gives you the fastest application startup time in the environment that you care probably the most about. And now think about this for a moment, what an environment is like, even with a really fast internet connection with a lot of bandwidth. If you don't have a mirror, every single device is pulling images from Docker Hub and the more people you have, the more devices you have, the more trouble you can run into. So this is a really useful pattern, it may even be one of the first things that you try, like myself, with hosting your own registry. In fact, one more idea you might want to try on your home network, if you have a NAS somewhere than can run Docker containers, then you can set up a local registry for your private images, as well as a registry mirror. It's one of the easiest things to do is to set up a registry mirror on a NAS device where you have a lot of storage, probably a fast connection to your local network at least, and all the devices on your network can probably already get to it. So if I hop in here to my containers running on my NAS device, you can see I've got a hub mirror set up, playing the part of a local registry mirror. So now that you've got these ideas in mind, let's go set one up and let's take it for a spin.

Challenge: Setting up a docker-compose.yml for a Local Registry

So we'll be creating another registry, tear down anything you have left from previous demonstrations, and this time I want to set up the registry mirror using a Docker Compose file. So I've gone ahead and created in the course repo a mirror folder. You can look inside of that folder in the source code for this course if you'd like to just copy my Docker Compose file. But if you would like a challenge, I'd really encourage you, before you do that, to pause the video and take a moment and see if you can go ahead and create your own Docker Compose file to get a registry up and running. If you want a challenge, you can get your registry UI running as well, but that's not necessary, I won't even be using that myself here. Instead, if you could just get a registry and then come back and I'll walk you through how to set up first a registry in a Docker Compose file and then we'll morph that into a registry mirror. Okay, so here is my work thus far that you can access out on GitHub under the course repository in the mirror folder, grab the docker-compose. yml. file. Now there might be a few more files because this is before I'm done with this course, but focus just on the compose. yml file. And here is the contents of that file. One service defined at this point in time. The image is the registry image and whatever is the latest version 2 image. Of course publish a port and then map in a named volume, a registry named volume, it will actually be mirror and then -registry, and then map that into the location where we are storing our images. And then last up, this is a new option we didn't use before, it can be nice to set a restart policy so that if for some reason the container dies or you reboot your machine, you don't have to go start the registry back up. So in this case, I like the option of go ahead and restart, unless, of course, I explicitly stopped this registry. And then because I have a name volume, I've defined that down below. Alright so let's fire this up. Quick quiz, how do I do that? Well first up, get inside of the folder where that compose file lives and then this is a simple call to docker-compose and then up, and I'll go ahead and attach to the containers to view the output here. Ignore these early warnings, after which you can see we are creating our mirror registry and then attaching to it. And then we have our registry container output down below. A few warnings about redis and an HTTP secret, we'll get to that later. The important part is that you should have a message that the registry is listening on port 5000, if everything was successfully. Just like you would have if you're using Docker run. So this is now up and running, what could I do to test my registry? Well first up, I cleared out the screen and I'm going to split my terminal. I'm going to use curl to make an API request and make sure the registry is running. What can I type in, and you could use a browser if you want, no need to use curl, but what address do I need to look at to check maybe my list of repositories? Well in this case, localhost:5000, or wherever you have your registry running, v2/ and then \_catalog. You can see right now we have an empty list of repositories. Also, if you look above, you can see in the output that our registry has logged that somebody has requested the available repositories. That's not important though, so I'll clear that out again and I'll clear out the bottom now, and join me in the next video where we modify our registry to make it a mirror.

Using an Enviroment Variable to Enable Mirroring

Alright so we have our registry up and running, but it's not a mirror yet. Now there are a couple of ways we can configure our registry. Do you want to take a guess what they might be? Environment variables and a config file are the two common approaches and that's pretty typically regardless the application you're using. Now the easiest is probably an environment variable right now. So if you come into your compose file and add an environment section underneath your registry service, and then if you add REGISTRY\_, this is the prefix that's used for all registry configuration via environment variables. And then there is a proxy configuration section, inside of which you can set a value for the remote URL to proxy, so what register would you like to proxy or cache or mirroring is a better word, I think. And then we just go ahead and paste in https, secure is a good idea, and then it's registry-1. docker. io. So this is the only line that you need, the only piece of configuration necessary to turn on this mirroring or caching mode. It's actually referred technically as a pull through cache, and that has to do with the fact that we ask our mirror for an image, it goes out and gets it and then brings it back, caches it, and then passes it through to us. So it's like we're pulling it through our proxy or our mirror. Alright, so make sure you save that. And now what do I do? Well we need to recreate or restart our registry to pick up this change for the configuration, and I'm a little bit paranoid when I'm working with some settings like this, and when you switch to a mirror, that's kind of a drastic change, it's a bit different than being able to push images to a local registry. So I personally like to just wipe out the registry if I'm going to fundamentally change the role it's playing, and that includes the volume that we created. And to do that, there's a docker-compose down command, it's symmetric to up, and if you grab the help for this, you'll see that there's a few flags you can pass, notably a v or volumes flag, which will get rid of named volumes. So just in this one case where I want to wipe it all out because I'm fundamentally changing how I'm using this registry, I'll bring everything down, and then actually I'll go up above where I was attached and watching the output, and I'll bring things back up. Alright, so what could I do to confirm that we actually wiped out our registry? Well, we could run a curl call here. Take a look at our repositories and our array here is empty, so that's a good thing. And if you look up above, when the registry started this time, there's a few more bits of logging or output here, and the notable log that you want to look for is this entry here that says that the registry is configured as a proxy cache. It'll give you the URL that you set up to the registry that's being cached or mirrored, and if you see that, everything's probably good, as well as, I suppose, having this listening on port 5000. Also up above, you'll see a new entry versus just running a private registry, so mirrors have an expiration scheduler. This is part of getting rid of cached objects. Alright so we have our mirror up and running. Join me in the next clip where we talk about how to use this.

Configuring the Docker Daemon to Use a Registry Mirror

Alright so I've opened another tab here and I've listed out a particular image that can be problematic, it's the Node. js image, the latest tag. And you can see it's 677 megabytes. This is the uncompressed size on disk. Over the network, this will be about 260 megabytes, so still a large image. Now depending on your internet connection, while you're tinkering around with a registry mirror locally, you might want to pick something smaller if you have a slower connection to compare speeds. So I'm going to start out by getting rid of this Node. js image from my local machine. So if I list out my Node. js images again, we have nothing here. And if I run a curl again against my registry, we have nothing cached. If I go ahead and pull the node image, what do you think is going to happen to that catalog of repositories if we query it again? And what do you think we'll see over on the registry output? Well let's run this and find out. And actually let me cancel that. Let's time this download, so I'll use a time command, which is a crude way to measure the time it takes because that will include both downloading and extracting the image. Nonetheless, most of the time is spent downloading, and there you go, we're done. By the way, for accurate timing, make sure that all of the layers in the output say pull complete. If any of them say already exists, that means you already had part of the image and you're timing is going to be inaccurate. You'll need to hunt down the parts of the image that are shared with other images and get rid of those other images or you might want to just find an image that you don't have any layers for if you want accurate timing. So this is a pretty fast connection to pull a large image in 13 seconds. Don't get too jealous because I still don't have that Fios gigabit connection that I really want. But I do have gigabit down, so hey, I've got half of the pie. Anyways, this is why I picked a big image because, as you can see here, 13 seconds, that's going to be hard to beat, and that's pulling directly from Docker Hub. I had a question though a moment ago, I asked what do you think will happen to our registry when I do this? Remember we had an empty list of repositories before we started. So let's clear out this screen and we'll run our web request again. And there's nothing there. By the way, remember you can run this request in a browser if you don't want to use your command line. I'm just kind of lazy here. Anyways, no repositories, why is that? Why didn't we use our mirror? Well I didn't tell my Docker daemon that I created a mirror, so how does it know to use it? Just because it's on the local computer doesn't mean it's going to sniff it out. And imagine if my Docker daemon was another computer, like if I went over to my Windows machine, obviously it wouldn't know what's going on on my Mac machine here and might not even be able to see what's going on if I have a firewall up. So we have the configure the daemon to use this mirror. How do you think I go about doing that? Well I need to configure the daemon and on Docker for Mac or Windows, that means going to the little tray icon and opening up the Preferences and then on the Daemon tab you can configure the daemon. Now there's insecure registries, as we used before with our insecure registry that we set up earlier in the course. There's also a section for registering mirrors. And as you can see an example as a placeholder there, I'm going to paste in the registry mirror I have set up, it's on plain HTTP, not S. And then I have pointed out my external IP address on my internal network for this machine, just so if I copy and paste this to other machines, I'm not so confused. I could have put localhost here, and then, of course, port 5000, whatever port you publish on. Since I've added that, you can see the Apply and Restart button pops up, so hit that. That'll take a moment, so maybe go grab a coffee or a quick snack, wait for the little light to turn green again down here in the lower left. If you're inpatient like me, you can go run docker ps in the command line. Alright, there we go. So we can come back now and what could I do to check to make sure that that configuration has actually been applied? Think back to very early in the course when we set that other setting, what did we do to verify it? Well before, we used docker info, now I'm going to use docker system info, the exact same thing, and then look down at the bottom at Insecure Registries, right below that are our registry mirrors, and you can see we've got our one entry listed. So things must be configured. So next up, let's pull down an image for the first time and take a look at what that's like through our mirror.

Pulling an Image Through the Mirror for the First Time

Let me clear this out, run my curl again, no repositories. Let's do our image pull again, Node. js, and let's time it this time. So remember, this time our mirror will not have the image yet. So it's going to have to go get it from Docker Hub for us. Now knowing that, what do you think will be the implications as far as timing? So this is where you will be penalized potentially, depending on how fast your internet connection is, because adding in an extra layer of indirection will slow things down a bit, and with a fast internet connections like I have here for downloads, chances are this will be worse than the 13ish seconds we had before going straight to Docker Hub. So let's run this and find out. Actually I need to remove the image I have first, and now I can pull the image and time that. Otherwise, I'm leaving the image in my local cache for my Docker daemon and, of course, that's not going to take any time at all, not unless somebody published a new version of Node. js. Doo, doo, doo, doo, let's see how, let's see how fast we are now. Alright we are done now and this time, uh, 29 seconds. So this definitely doesn't help me out. But you probably aren't always, and I'm not always, on a fast internet connection, and there are some days where it's just not as good as it is right now. And if that's the case, this extra layer may be negligible, especially if pulling an image this big might have taken you 10 minutes because of your internet connection, well adding a little bit of overhead on top of that, that's not a big deal. Alright now let's run our curl. Let's look at our list of repositories. Drum roll please. Hey, look at that, we've got a first mirrored or cached image and that means we should be able to take advantage of that now to quickly download that image onto any machine that uses this mirror and doesn't have the image yet. So let's try that out in the next clip.

Even with Gigabit Download Speeds, a Mirror Can Help

Alright so now that we've downloaded this image and it's pulled through our mirror cache and we have it cached locally, as we can see here, let's go ahead and blow it away again with an image rm, so blow away the node image, and then let's go ahead and pull it again. And of course, let's run some timing on this. And what do you think this is going to look like? More than, less than the last two, or somewhere in between? Let's find out. And there you go, 12. 877, so it's not much. And I did happen to keep the previous two runs. I copied the values, so let's put this up next to those. So this last run here is the second pull. Not a big difference here, again, I'm on a fast connection. In a moment we are going to throttle our connection and see what happens, but 12. 877, that is still faster than the 13. 7 that I had directly to Docker Hub, so we do have an improvement here, even on a gigabit connection.

Timing Docker Image Pull on a Slower Connection

Okay so we've been trying things out here on a single machine and we've seen slight improvements, but let's see something real here that makes this a little bit more relatable so you don't have to imagine so much, which I think is a good thing when we're learning. And while I'm at it, how about we give Windows some love here. So I'm over on my Windows machine and I said that one of the big benefits is when you have multiple machines on a network. And it's in this setup that you can really benefit from having a registry mirror, especially if you pay for the bandwidth that you use and you don't have an unlimited amount, or if you just don't have that great of a connection. Now before we get our Windows machine set up to use our mirror, let's throttle our internet connection just a smidge and see what the difference is like when we're pulling an image. It just so happens I have a smart switch in my office, and Port 1, which I've checked the box next to here, Port 1 is where the traffic goes in and out that eventually gets to the internet. So my uplink connects up to my router somewhere else in the house. Anyways, it's unlimited right now, but if I apply a limit to it of about 20 megabits, than that's definitely going to slow things down a bit. Now nothing wrong with a 20 meg connection, don't feel bad if you're there or below. I travel a lot and a lot of my family is only at 1 or 2 on a good day. So now we're going to pull that image, and if you look out on Docker Hub, you can see the size of what we're going to pull under the tags for the Node. js image. You can see the latest tag here is about 270 megabytes. So the 677, that's actually expanded on disk, this is the size roughly that we'll be pulling across the wire. Okay so back on the Windows machine, with this throttling in place, first I want to just make sure that I'm not using the registry mirror. How do I do that? So I can do docker system info here, and down at the bottom, I shouldn't have anything under mirrors. So we're good to go. And just to be safe, let's make sure we remove the node image if we have it already. Okay. We can always list out our images, and you can see we don't have that image right now. Also, make sure you're using Linux containers if you want to compare apples to apples. So if you look at the popup menu for the Docker tray icon, you should see Switch to Windows, and that'll indicate to you that you're using Linux containers, or look in the output there from the docker info command. Okay so a first pull here of the node image, and I want to time this, but with PowerShell to be a bit different, we'll just look at command history after the fact to produce the timing info. So let's do our first pull here and a throttle connection directly to Docker Hub, not through our mirror. As you can see, this is not as fast as what we had before on my Mac, and that's thanks to our throttling. So I am going to go grab some coffee or do something quick and come back here and we'll take a look at the timing information. Oh I can't resist, forget the coffee. Does this feel like déjà vu for anybody? This reminds me of sitting in the basement of my sister's house trying to get some example ready maybe for a workshop I'm going to teach or a course I'm going to teach and being so incredibly frustrated because the download speed is just atrocious, especially on those last few layers that are big. Alright so we're done here, now we can calculate the time by going back into the history command and grabbing the last history entry and looking at the end versus StartExecutionTime of that last command. So what PowerShell has is this nice object oriented shell and we can use that to produce the TotalSeconds then, and we're at 121, so just shy of 2 minutes. So the throttle clearly has had an impact on simulating a slower connection.

Configure Docker for Windows Registry Mirrors

Okay so let's clear this out here and let's set up our registry mirror on our Windows machine. How do I do that? Well I just need to hop into Settings, like on my Mac, go to the Daemon, and then down below I have registry mirrors, just like on my Mac. So I'll paste in my registry, and make sure you have your firewall opened up to that mirror, wherever that's at, and then apply those changes. Then get a cup of coffee again. Alright green means go, which means run a docker system info. Hey look, I'm a poet. Alright we've got our registry mirror added. What should I do next in this text case? Well it's tempting to pull the image, but we still have it, so we need to get rid of the image. There we go. Let's list them to be safe, and let's pull now. Alrighty, looking better. Alright, we are done and, yes, I sped that up. Before we look at the timing information, I want to reiterate that part of the time is for download and part is for extraction. So this is by no means a specific comparison of just download time. Though that definitely felt faster, let's look at the timing here, and hey, look we've got 37 seconds, much better than 120. So as you can see, when we pull through our mirror on our local network, that's not throttled, it's just the uplink that was throttled, you can see we have much faster access to our images then. And this is a valid assumption, your internal network is going to be a lot faster than your connection to the rest of the internet. We've now reached the end of this module. I hope you can see how beneficial it can be, especially if you have a lot clients pulling from Docker Hub, to instead spin up your own registry mirror internally and redirect all of your traffic to your registry mirror. So this is another benefit or another use case you can add to your list when it comes to self hosting a registry.

Automating Builds with Notifications

Automating Image Builds with Notifications

At some point as you continue to adopt containers and images more and more, you'll realize that it would be nice to have some automation around the process of building new container images and then deploying new containers based on those images. And one of the nice things about a private self-hosted registry is that you have a lot more control over that registry, and that control tends to afford a lot of flexibility when it comes to automating the tasks that are a part of developing and deploying software. So let's take a look at what that flexibility might look like. So let's say that you're using the NGINX application inside of your organization and naturally from time to time the team that's responsible for NGINX is going to release new builds of NGINX and ultimately at some point we will see a new image with a new version of NGINX inside of it and that will be pushed out to Docker Hub, and that's a beautiful thing because then we can pull down an entire environment that's perfectly safe and vetted by the people that wrote NGINX. So we know we're starting from a good foundation. Now perhaps your organization is security conscious and maybe you have some sort of team somewhere that wants to vet images before you use them. Even though this might be an official image that's sponsored by both Docker and the team developing NGINX, you still might want to put that image through some testing before you allow it to be used inside of your organization. And maybe that's taken care of by some sort of security team. Well one really good use case for a private registry is then to set up a security registry that's owned by that security team that contains vetted images. And once an image has been vetted after a new release occurs of any given piece of software that you use, that image can then be pushed out to that private registry and then it can be subsequently consumed by other people inside of your organization. And maybe, for example, you have another team inside of your company that's tasked with educating people about how to use software. So for example, they might have a program to teach people about NGINX. And as a part of the learning program, maybe some additional files are helpful, maybe website files, and it's desired to add those files into an NGINX learning image so that when you're running an NGINX learning program, you just have the image ready to go, people can spin up this learning environment with NGINX inside of it and all the files, and then off they can go focusing on learning and not on setting up the environment to learn inside of. So that team of people that create the learning programs can pull that vetted image so that they're never encouraging people to use an image that hasn't been checked. They can pull that image down, make some customizations to it, and push it, perhaps, to their own private registry, perhaps a learning registry inside of your organization. So another thing you can do with registries is just have multiple of them that belong to different teams of people, and then each team can do as they set fit with their registry and anybody else in the organization that they would like to grant access to can pull images from their registry. For example, a student that comes in to take that NGINX program and learn about NGINX can then pull down that image that's been both vetted and has the additional exercise files inside of it, to get up and running quickly with NGINX. So this is a hypothetical situation. I hope it gives you some ideas for maybe what you might want to try to do with your own private self-hosted registries. With this scenario in mind, what I would like to do in this module is focus on that automation that can make this scenario more of a reality without the hassle. I mean, could you imagine if every step in this chain were a manual process? That'd probably be a pretty big hassle and then it might be something that people just don't do. So in this module, I want to drill in to this middle part of the chain, after an image has been vetted, when it's pushed to the security registry of vetted images. I'll show you in this module how you can use registry notifications, which are webhooks, to trigger a process then that could kick off an automated build to add in the exercise files for learning purposes. That way every time a new vetted NGINX image is pushed out, you can quickly create a new learning environment. That way, when people are learning, they're using a vetted image with the latest version of the software. And of course, the sky is the limit with what you could accomplish with notifications. So as you work through these exercises in this course, be thinking about how you might put this into practice in your own work.

Building a Mongo Image with Sample Data

So here's the example we'll work through in this module. Instead of NGINX, I have a slight permutation, I have an image that will be built, that will have a MongoDB server running inside of it, and it's based on the official mongo image. This is published out to Docker Hub and we could hypothetically have this vetted, once vetted it'll be pushed to a private registry, a vetted registry, for that part of the demonstration we will push it to the private registry that we've set up and we will see then how we can kick off a build process that will add in some sample data to our MongoDB database so that when students pull that MongoDB database down and spin it up out of that container image, so when they start up a container, they can get right to work learning about MongoDB, they've already got data loaded, they don't have to worry about any of that, they can just start exploring querying and interacting with the data in a MongoDB database. Which is a great way to get up and running. Just imagine if every single time you wanted to learn something new, what if somebody already set up an image, maybe even out on Docker Hub, or maybe internally in your organization, that had not only the tool, but also a playground with some sample data for you to interact with? Wouldn't that be a wonderful thing? For this particular example, I have a separate repo from a long time ago that has an example of Docker with a sample data set built into an image. Make sure you select the branch called docker-registry. I've cleaned this up just a little bit to simplify things and make it work for our scenario. And down in the README, if you want to follow along, I have quite a few of the commands documented here so you can easily follow along. I've already cloned the repository to my computer, you can see the four files here. So first up, let's just talk about the Docker file and see how that works.

Setting up the Mongo with Data Example

I have loaded up the files for this repository into vs code so we can walk through this. The Docker file contains a multistage build. The first stage downloads the dataset and then the second stage assembles our runtime mongo server environment, copying in notably this sample entry point script that will load up the sample dataset that was pulled down in this previous build stage. So at the end of the day we'll have a mongo server that comes out of here that has an initial dataset that we can interact with. So that's one piece of this example. The other piece is the docker-compose file. This basically spins up four separate containers for two separate environments. One environment, just so you can see what it looks like, is a mongo server with no data, and then I have a GUI web app called RockMongo that looks at that mongo server so you can see there's no data, and then the separate environment here down below is our mongo image that has our sample dataset in it, so we spin up a mongo server with data, and then a separate instance of a RockMongo web UI that points at the mongo server with data so we can see what that looks like and compare and contrast the two. Long story short, this will map or publish to ports onto your host 8090 and 8091 and those are the two things you can interact with to start playing with the data and start learning about mongo. So we have an entire learning environment packaged up here. And you can quickly spin this up to follow along. Over at the command line, run docker-compose and up, and that's all you have to do to get this entire environment created, as well as to build the image. And you saw a moment ago there the build kicked off as a part of docker-compose up. So the image is now built and we have this environment set up or actually we have this set of environments set up, and if you come over to a browser, if you first pull up localhost 8090, you'll hit the RockMongo GUI web UI that has no data, and if you look over to the left you will not see our text dataset that you'll see in a moment here. And then if you open up a new tag and go to 8091 instead of 8090 you will see our test database listed and if you click on that, inside of there is a restaurants collection, inside of which you have documents that you can interact with, and this is RockMongo where you can fire off some queries and see visually the result of interacting with a mongo server. So that's the difference between these two environments, there's no test dataset over here. If you're not inclined to learn from a GUI, you're welcome to go over to the terminal, open up a new terminal, and you can use docker-compose exec to exec into either of the mongo servers, so that would be mongo\_empty or mongo\_with\_data, and those, again, are the names from the docker-compose file. So you can see mongo\_empty and mongo\_with\_data, those are the names of the respective mongo server services. That's where that comes from, and then you can open up a bash shell. Once you're inside, you can use the mongo shell, just by typing in mongo and return, and inside of here, I'll clear this out, you're welcome to use the test database, you can even show the databases, you can see the lists that we have over in the RockMongo GUI. You can show collections. There's our restaurants collection. And you can do a db. restaurants and they maybe. find here to dump a list of the documents to actually open up a cursor in MongoDB, and you can iterate through the documents. And if you want to try out a few different commands, you can come over to the README for this repository in that Docker registry branch and you'll see some commands you can fire off to learn a little bit about mongo. Feel free to take a tangent here. This ability to easily learn with Docker and containers and images, which is the impetus of this module, is extraordinarily powerful. As you can see here, if you've never worked with mongo before, well, hey you've got an environment set up here in just a few minutes where you can start to learn about mongo without needing to know how to set it up and install it and without even needing to know how to load up sample data. I think that's pretty cool. So now let's talk about how we can kick off an automatic build whenever MongoDB upstream changes and we want to build a new version of our dataset into the latest version of mongo. Let's talk about that next.

Setting up a Registry Config File

Now that we've seen the image that a student will use, let's take a look at the process to build that image and see how notifications in a registry can help us kick off that build process automatically. We'll be using Jenkins to do that. So the demo will set up, we'll use a vetted registry that'll have a configured notification that will tell Jenkins when a build needs to kick off to go ahead and add the sample data because we have a new security vetted mongo image. Once Jenkins is done then, it will push out to the learning registry and thereafter any student can use the image. So the first thing I want to set up is our vetted registry and I want to create a configuration file so that we can then set up a notification. Now here's a sample notification in the documentation for the registry, and as you can see, this is a yml type format. This will go into a config file that we'll set up in a moment, and that config file serves as an alternative to using environment variables, as we saw previously with the registry mirror. So two different routes to go to configure your registry. Now the best way to get your hands on some sample files is to hop over to the location of the Docker image Dockerfile for the registry. You can always find this by looking out on Docker Hub, there's usually a link inside of a Docker Hub repo, this will link back to the Dockerfile, and in this case you can see a few copy instructions, of which the second copies in a config-example file and it's copying in from the registry directory, next to this Dockerfile, and then this config-example, and it's a yml file. It's copied it into the image right here, so you can use that, or you can come up a level here, go into registry, and go into the config-example. yml file. You can grab a raw version of this file and just paste it into a yml file on disk. So that's one route to go. Another way you can go about this is to just copy the file right out of the running registry. So for example, I have that mirror up and running still. A simple call to docker container cp allows me to copy a file out of a running container. I just have to specify the mirror registry, and then I need to specify a location inside of the registry, which I copied from that Dockerfile. Go back here, I'll show you that again. This is the second parameter for where the file is copied into the image. So I've pasted that here, so I'm copying out of the mirror registry container this particular path, which is a file, and I'll copy that into that current directory, which happens to be this course/mongo folder. So I've created a new mongo folder to work on this notification example. So running that command then, I can list out the contents of the directory and you can see the config. yml file. And if I open up this file, this should look familiar, what we just saw out on GitHub. So this is a starting point for a configuration file that's used by the default registry image. So there are some reasonable defaults here, some of which we'll explore throughout this course. For now though, we'll ignore most of what's inside of this file and we'll turn to configuring our own notification next.

Configuring a Jenkins Notification Endpoint

Now in the process of making changes to this file, it can be nice to check it into version control. So the very first thing I'm going to do here is add this new config. yml file into my Git repo. That way I've got a clean, working slate to start from. I'd recommend you do the same, that way if you change any settings, you can undo them easily and you know what you actually changed. This is incredibly helpful if, for example, you break your configuration file that was working a moment ago, you can quickly revert instead of pulling your hair out for half an hour. Alright so inside of this configuration file, there are various different sections and one of the sections that we can add is a notifications section. Inside of here, there's a list of endpoints that we can add and these are endpoints that the registry will call out to, so webhooks is another term you might have heard of. Basically an HTTP endpoint that the registry will ping and send some data to, when certain events happen with inside of the registry, for example, when somebody pushes a new image. And specifically it's an event around pushing a new image manifest. We'll see that in a moment. For now we just need to add our very first item to the list of endpoints, and the first thing we'll do is add a name to that endpoint. And in our case, we'll just call this jenkins. Of course, in your own environment you'll want to be a little bit more specific about what this is, especially if you have multiple endpoints. That can help somebody else troubleshoot what the heck this endpoint is for. So if this is an endpoint that calls out to Jenkins, what's another piece of configuration that we're going to need inside of here? Well, a URL is a likely candidate. We'll need to call out to wherever Jenkins is located. So wherever you are Jenkins. We'll come back to this in a moment when we set up Jenkins. And don't worry, if you don't know how to set up Jenkins, I have a repo you can pull that will do all the work for you, of course, using Docker containers and images. Alright that's enough to get started with configuration, let's save that and then next up, if you'd like a challenge, go ahead and create a docker-compose file to boot up a new vetted registry and see if you can figure out how to set up this config. yml file to map in and be used instead of the one that comes inside of the image. So that's if you want a challenge. Otherwise, join me in the next video when you're ready to see how we set this up.

Creating a Registry Container That Uses the Custom Config File

Okay so here is our compose file that will set up a new service for a vetted registry. We'll run this separate of our mirror that we set up before. And in order to do that and access the registry, I've mapped a separate port here, 55000. So that's mapped onto my host. Make sure you have a firewall on your computer, otherwise this port when you're publishing it, as always with Docker containers, will be accessible to the world and that may not be a good idea, depending on the network you're sitting on right now. If you want, you can prefix this with 127. 0. 0. 1 and then that will only bind to your loopback or localhost address and not to the world. I won't do that though. I'll save this file and then I'm using the registry 2. 6. 2 as before, and the important part here is the volume mapping. I'm taking the config file that's sitting right next to my compose file and you can see that on disk here, you can see the docker-compose and the config-yml file sit right next to each other. So I'm mapping that in over the top of the one that's inside of the image. And that's all I have to do to boot up a new registry that uses the configuration file that I have provided. So let's try that out. So what do I type in over here to kick off this registry? So I'm sitting inside the directory next to the compose file, I can just do a docker-compose and then up, and I'll stay attached so I can see the output. When I run that now, if you look through the output there should be something new. Can you spot it? Well one of the messages in our output here says configuring endpoint jenkins. And you can see the URL that we've provided, of course, bogus. Nonetheless, at this point in time, if you get this message, that means that you've successfully configured your registry with your own endpoint, which means that you're now using your own configuration file and no longer the one provided in the image. So if you don't see this, you know you have a problem, take a look at things and see what's going wrong, and then let's get going with the rest of this demonstration. Now one tip when it comes to looking at the output here, there's a lot going on, I find it helpful to dif the output before and after I make changes to the registry configuration. Let me show you how to do that here. So I could kill this off and then I can make a new tag here, so I'll keep the old output and I'll have the new output over here. So I'm going to run a docker-compose up again. Before I do that though, I'll come comment out the volume mapping so I'm no longer using my custom configuration file. Alright, so now I can docker-compose up here. Alright, now I have the output without my endpoint configured, and if you toggle between the two tabs, you'll see that difference. Now if you want to kick this up a notch, you can put this into a dif tool and look at a line by line difference. Okay so let's kill that one off and go back to the original, and then let's undo the change to the compose file and save that. Of course, it's not just the compose file, you could have made a change to the config file itself maybe to comment out the whole notifications block. And of course, we'll docker-compose up again to make sure everything is okay and we should have that endpoint message, and that is our indicator that we're all good to go.

Configuring Json Formatted Logging

Now if I may, while we're on the subject of logging, I find this particular standard output, this combined logging format, to be somewhat hard to parse. The good thing is, we have a config file now and there's an option in that config file that allows us to control this logging, so now we can start to tweak those settings as well. So let's do that. I want to change over to JSON formatted logging. I personally find it more helpful and I say that because I'll show you how you can use jq, a command line tool that's cross platform, to parse that JSON and take out just what you care about and you can produce a really nice compact log then to look at instead of all this gobbledy gook here. Now if you'd like a challenge, why don't you try to change the log formatting to JSON, get that working and come back and I'll show you how to do that. Okay to change the output, come under the log section and then add formatter as a setting and set the value to one of text, json, or logstash. By default this is text, which over in the console is why we see the equal sign delimited field, so we have time= and then we have the value here. And then we have level= and then we have info and msg= and then this long string here. We also have the go. version and da, da, da, da, da. So text is the default. I'll set this to JSON and we'll save that. Alright with that change in place, I want to ask you, what do I do to get my registry to pick up this change? So it's important to distinguish in this case we changed the configuration file for the registry. We also have our docker-compose file and it's also a yml file, so sometimes this can be confusing. But if we change the config file for the registry, we need to make sure the registry process picks it up and for that we need to restart the registry process. Now there's more than one way to do that. Just keep this in mind, if you, for example, come over to a terminal window, if for example the registry is somehow running in the background, you started it previously maybe in the background, so it's a container on your system, so you can see that I have the registry up and running right now and I can dump out the logs as we saw a moment ago. Because I'm not attached to that registry process in any way, I'm not going to stop it if I run a docker-compose up. That's why you can see here, when I run up, again, nothing changes. I just see the previous log output, and as you can see here, this particular registry service is up to date, so there's no changes in the compose file and that's why nothing is going to change if I run the docker-compose up in this case. So I'll detach from that, clear out the screen here. So we'll still up and running. I was just attached to the logs there. And I'll pull the old log output here so you can see it. This time if I restart and then I look at the logs, now you can see somewhere along the way we switch over to the JSON format. So just make sure that you restart that container to pick up the changed registry configuration file. A shorter way to do all of that, to both see the log output and to make sure that the process restarts, in this case I like to just run docker-compose up and force-recreate the vetted-registry. I like to be specific about what I'm force recreating so I don't wipe out the wrong thing. This will go ahead and recreate the container not matter what, even if I haven't changed the compose file. So then this is all one step to then see the new output. And there you go, you can see we still have our JSON output. So just keep that in mind. Another benefit of this approach and why I also like it is because it wipes out the log history, so I don't see previous log output and become confused. Now on looking at the output you might notice that the very first line is not JSON formatted. This is a log message that comes out before the JSON formatter kicks in. This is not a big deal, but if you want to get rid of this to avoid confusion, this is simply complaining because we've not set a log level and it's defaulting to log level of info. I might as well point this out so you can see that you can change the log level here. You can turn up or down the verbosity. You have debug at the most verbose level, info right above that, and then warnings and then errors above that. Info is the default, if you want you can change this or you can set the default just so that you can see the property or setting in place and know that you can change this if, for example, you want to see more log output, or less. So you can save that change then. Hop over to the command line and I'll clear out the bottom screen here. I'll Control+C to kill off the docker-compose up where I'm attached to the container, which stops the process, and I'll just go ahead and force recreate the vetted-registry, just to be safe here. Now you can see that very first warning is no longer in place because we've set an explicit log level. By all means, not necessary, but if you want, you can change that verbosity to see a little bit more in the output. So next up is taking this and parsing this to make this a little bit more readable because even in the JSON format it's in, this still is not that easy to look through.

Cleaning up the JSON with jq

Alright so we've got a lot going on in the log, we can trim this down now that we have JSON, using a command line tool called jq. This is cross platform, I won't go into the specifics of it, I definitely encourage you to take a look at it if you're working with a lot of JSON data in the command line. It gives you a structure to parse through things. A lot easier than remembering regular expressions and working with text parsing, at least in my opinion. Nothing wrong with the text format if you like it, whatever floats your boat. So all I've done here is made a new tab so that we can leave our registry up and running, we already have the log output that we need. We can grab that with the logs command. And now we'll need to pipe this information through jq and to do that, to save some time, if you hop over to the repo that we're working on for this course, right next to the config file that we're working with, there's a README file that has some helpful commands inside of it. And somewhere inside of here, you should see some commands that show you how to use jq. For example, if you copy this very first command, and this is formatted for bash, jq is cross platform though, so this would work in PowerShell. You would need to change the grep portion of this. Now over in the terminal, I can just paste in this preformatted command, it's a multiline command. We just run Docker logs, following the vetted-registry, so we'll continue to see new log output. We'll run things through grep to strip off that service name that's on the front. So basically to strip off this whole part, we're looking for the JSON formatted payload, taking every basically inside of that formatted payload, and then I'm piping that through jq with no modifications. So let's take a look at the output. Now that looks a lot better. This is a lot easier to parse through and find, for example, here's the level, here's the message, here's the message, here's the message, this is easier to work with, in my opinion. And then there's a second version of this command just with different parameters to jq. This formats messages a little more compact. Everything's the same, just different restrictions on what I'm filtering through jq, taking just the three fields that I think are important, the time field, level, and then the message. This gets you almost a stacked like view. I have a pretty large font right now. You can see if I zoom out a little bit, it's a little bit more tabular and a little bit easier to work with. So use both of those to troubleshoot your registry output.

Adding a Standalone RequestBin to Inspect Registry Notifications

Okay so we have the basic pieces in place to be able to send notifications, but one thing I've found is it can be very difficult to troubleshoot problems with registry notifications, as with any sort of webhook. And one nice thing that you can have is some tool that'll allow you to just look at the payload of a notification, so the JSON that's going to be sent out of the registry each time a notification happens. It can be nice to see that with some sort of inspector, also like a debug tool, and a really nice way to do that is a tool called RequestBin. So quickly here, we're going to validate our registry configuration and notifications and learn a little bit more about the payload by setting up a second notification that we'll use throughout the rest of this module to help us troubleshoot. And for that, we're going to use a standalone RequestBin instance. I don't know if you've used RequestBin before or not, but it's basically a little website that allows you to capture web requests and then look at them. So it's a server you can stand up, and this is a standalone instance that, of course, we will run inside of a Docker container with a prebuilt Docker image. And then we'll be able to inspect the notifications before we even put them over to Jenkins, and then also while we're working with Jenkins. So first up, let me show you my modifications to my Docker compose file to stand up the second standalone RequestBin container. Alright so here's my compose file, it's this service right at the end, just a few lines of code, if you will, and we can have our own standalone RequestBin instance up and running. Now one thing, I built this particular image for RequestBin, I'm hoping pretty soon there will be some sort of official image as part of the RequestBin project, so you don't have to trust my image. Though for the time being you can use this, or maybe build off of my example in my GitHub repository called RequestBin. So you can look at this Docker Hub image and get into all the bits of this if you'd like to see more. Also another thing of note, I am using here a single container and the RequestBin instance inside of it will use in memory storage for storing the data, which means if this container goes down for any reason, if it crashes, if you restart it, if you just stop it, if you reboot your computer and forget that you have it running, you're going to lose all of your data. If, again, you look at my Docker Hub repo for this, you can find instructions for how to set this up with Redis, a few more lines in this compose file and you can have that up and running and then have some persistent storage. Okay so first I want to get that RequestBin instance up and running. What do I type here? Well in this case just docker-compose up again, and to bring it up alone in its own separate window, just type in requestbin. And then if you come over to your browser, localhost port 8000, or whatever you publish to, you'll see RequestBin up and running. From here just click the Create a RequestBin button and you'll have some instructions for how to use this. There's a curl call if you want to test it out. For now all we need to do though is copy this URL and then what am I going to do with this URL if this is where we send requests to? Well this is where we're going to send our notifications, so we should hop over to the registry config file and then let's come down here and just add in another endpoint. We'll call this requestbin. That way we can keep using this even when we start to set up Jenkins. If we have a problem, we can hop right over to RequestBin and see the exact same data. Now I just took off localhost here because of docker-compose, I need to use something else. What do I use here for the URL for the host name to send a notification to RequestBin that's now a part of my docker-compose file? What do I type in there? Well we can take advantage of the shared networking with docker-compose and use the requestbin service name. We can put that right over into our URL for our notification. So just to clarify, the vetted-registry will talk to requestbin via the docker-compose networking. And then we also have the little slug here for our RequestBin, this will change if you reboot that container and lose your data. So try to avoid that. But since we're just doing one off testing, the in memory storage of RequestBin is fine. So we'll save this file and now what do I need to do? Well in this case, let's just hop over to the terminal and we need to restart the registry. So I could come back to the first window where I have that opened up at, just kill that puppy off, and I'll force recreate just the vetted-registry. Do not recreate your RequestBin server, otherwise you'll have to generate a new RequestBin. So let's go ahead and recreate our registry and then I should do something here before I look anywhere else. What should I do here in the output? Well I should look for the endpoint that we added. Yep, there we go, we've got jenkins first and we have requestbin second. Now to test this out, join me in the next video for that.

Testing a Push Notification to RequestBin

Once we have a notification set from our registry, when you refresh this page, you will see it show up. If you continue to see instructions here, that means that RequestBin has not received any notifications or requests. You can test it out with this request right here or take a look at your configuration if you're not seeing anything come through, but for now we haven't actually sent anything, so let's trigger a notification to come through. And the best way to trigger a notification is to either push or pull an image. And considering that we do not have any images in our registry, our vetted-registry, yet, we only have one choice and that's to push an image. So for this, I'm going to use the busybox image, I've already pulled that down. Now can you tell me what I need to do with this busybox image to push it to our vetted-registry? Well I need to start by tagging that image, by giving it a new name, and the name needs to contain the registry that I want to push to, so in this case it's going to be localhost and 55000. Well you tell me, why is this localhost:55000 here? Well that's what we used over in our docker-compose file. That's the port we published on to our localhost interface to access our vetted-registry. Alright so once I've tagged that or named that, now what do I do? Well I need to push it, so a simple call to docker push here with the new name and I actually already had done that in preparation for this demo, apologies. Your output will look a little bit different as the layers move out, but it doesn't matter, if I push this, I'll have triggered what I want. So over in the RequestBin instance, I can refresh now, and take a look at that, we can see the payload of our push notification or the notification with regards to our push. And there are actually several notifications here, the first one of which is a push of the busybox repository, followed by some pulls looking for the various different layers, you see that in the blobs here. So there are actually multiple notifications that come out when you push an image. I would encourage you to refer to the documentation to see what's inside of those various events, to figure out what you might want to do with that extra information. Then nice thing is we can see the information here in our registry when we push an image, and we can use this now to troubleshoot anything that might be going wrong. Just the fact that we're getting this request to come through to RequestBin is a thumbs up, it's what we're looking for right now to make sure that we have everything in our registry set up okay. Because this is going out, then obviously it's possible for our Jenkins notification to go out. So let's talk about that next. In the meantime, if you want to, you can hop over to the vetted-registry tab, take a look at the output here, there's actually a lot of output, you might use jq to parse through this. So we can use jq here instead. There's a lot of messages coming out here and one of which I wanted to point out quickly is this registrysink: error. Jenkins does not yet exist, so the registry is freaking out, well it's not freaking out, but it's trying to repeat the sending of the notification in the case that this was just a simple failure. That is why we have some of those settings for the notification, for back off and timeouts, because otherwise the registry is going to pretty aggressively try to resend the notification, which is a good thing if you just have a small blip, but could be a bad thing when you don't have your Jenkins server up and running, you'll start to get a lot of output here and that could become very confusing to you. so what I would recommend is to just go ahead and kill off your registry for now, killing this will effectively wipe the slate when we recreate the registry here, so that we don't have that notification to retry anymore. Now when I perform this recreate, I want to pass an additional flag to wipe out the anonymous volume that's back in our registry, that way I don't have any previously pushed images, I have a clean registry each time when I run this recreate command. And for that it's a capital V. If you want, you can grab some help for this, it's a rather recent addition to docker-compose. So it's this new flag, -V, to renew anonymous volumes. So let's go ahead and close the help and recreate our registry with an empty slate.

A Fully-bootstrapped Jenkins Environment

So we're going to set up an environment now where we send that notification to Jenkins, in addition to RequestBin, and Jenkins then will kick off a build process that will build our new mongo image and push it out to our learning registry whereby students can pull down that new vetted registry with mongo and with the mongo sample data. Before I proceed to set things up here for this last demo, I want to say I've reset everything on my particular machine. So all the previous demos, I've wiped out all the containers and volumes and images for that, I've actually switched over to my laptop computer. So go ahead and tear everything down if you haven't done that already, and then proceed with the steps here. First up, if you go out to the course repo, I have a special branch that tags the starting point that I'm at right now. And if you take a look at the remote branches, you'll see a grand-finale-notification-challenge branch. Take that and check that out. And then make sure you're inside of this mongo folder here, and if you look at the contents, you'll see a few things, some of which we already had in previous clips in this module, for example, the docker-compose file. We also have now a learning-jenkins folder, which has all the bits to get a Jenkins instance up and running, and a recreate script to make it really easy to start things up. So let's take a look at these changes with the code editor. If you expand out the learning-jenkins folder, you'll see a Dockerfile inside, not too much here expect an inclusion of this Jenkins-bootstrapped Docker image that I've created. You can go look at that on Docker Hub and it links out to my GitHub repository for this, if you'd like to know the inner workings of this particular image and what I apply to Jenkins automatically, but the TLDR is I boot up Jenkins, skip the setup wizard, and just provide some default security to lock down the instance with an admin user and a hard coded admin password. And then I have an additional line in this file to copy in any additional files that you would like, one of which I've included for a job. In this particular job, we'll build and push our image. So you don't need to set up the job then for this example. If you know how to use Jenkins and know how to configure it, you can drop any files that you would put into your Jenkins home directory into this ref folder and they'll be copied in and they will be then applied to your instance. If you're not a Jenkins guru, that's okay, just ignore all of that, the long story short is this folder here has everything necessary to boot up Jenkins with a job to perform our build of our image and push it out to our learning registry. I also have this docker-compose file with a few additional services inside of it. I have added a second learning-registry, this is on top of our vetted-registry, and this simulates the reality that you might have multiple teams of people with their own registry that they control, and that's another nice use case for registries. Now in the particular case of the learning-registry, I have not exposed or published any ports back to the host like I did with the vetted-registry. We'll exec into the container for that if we want to take a look at the contents of it, we don't need to see that, but if you'd like you could go ahead and publish a port for that. Also, the learning registry just uses the default registry configuration, no special config file is getting mapped in. If you scroll down, you'll see our requestbin instance, just like before, and then I also have one last service, I'm running Docker inside of Docker, so that in our little Jenkins world, we'll be completely decoupled from the host machine, and this really simplifies this demo and makes our life really easy because we don't have to worry about configuring our host to access all these registries and going back and forth between Jenkins inside of a container, back to the host. We don't have to worry about any of that, we just have our own little Docker instance running inside of Docker for Jenkins to use. And I've preconfigured this to work with the two separate registries, so it's a nice explicit configuration of everything here to start up a Docker instance that Jenkins can use that can work with both of our registries. And then way back up at the top we have the last service here I skipped over, which is Jenkins. Setting the build context to that learning-jenkins folder that we just looked at. The Dockerfile is inside of there. I also have a named volume I've set up for the Jenkins home directory, it's not necessary, but this makes it a little bit less likely that I'll blow away this volume when I'm working with docker-compose. I have published a port for Jenkins and I've only bound this to the loopback interface for security reasons. I don't want anybody from the outside world to be able to access this, and that's mostly because I have that hard coded admin admin user and password, which is not that hard to guess and I don't want people setting up their own jobs that run things willy nilly on my machine. Even inside of a container, Jenkins can have access to a lot that might not be such a good idea. Last, I've set up an environment variable for the Docker host for Jenkins to access and this is configured to point at the Docker in Docker instance. Alright with that understanding in mind, let's hop over and boot up this environment.

Starting Two Registries and Jenkins

So back over at the command line, we have that recreate script that I would like to use, and it's really just calling a series of docker-compose commands together. It's going to wipe out the environment with down to begin with, that way you can call this repeatedly as you make changes, if you want to just blow everything away and recreate all of your containers. I also have this set up to wipe out volumes, both named and anonymous. And then I go ahead and perform a build with an explicit pull, followed by starting everything up in the background, and then immediately attach to just Jenkins to look at the logs for it. So all I have to do is recreate things. I will speed up the recording here for you. It looks like Jenkins is mostly up and running, so let's hop over to a browser and go to localhost:8190, that's support that I published for Jenkins, and use admin admin to get in. Once you're inside, you can see the docker-mongo-sample-data job. And then over on the left, I'm going to open up Blue Ocean, it's an interface I like to look at for simplifying Jenkins, and I'll drill into this particular job here. You can see it has not yet been run and that makes sense, we just started it up. And then what I'm going to do is open the configuration for this job in another tab so we can refer to it. So just a quick step through this and you can see this all inside of that job file if you'd like. I have set up a build trigger. This will receive the webhook notification from our registry when a new mongo image is pushed out. And I actually have the definition of several repositories set up, just so you can see you can define various repositories, multiple of them, to trigger a particular job. Down below then, this particular job uses a pipeline script. I tried to keep this simple enough that anybody could understand this without being a Jenkins expert. And I have some diagnostic in here to print out environment variables and then pull down a Git repo with the docker-mongo-sample-data sets and, in particular, I'm grabbing this branch right here that I have set up for this demo and that I will leave alone for this demo so that you can run it in the future. I run a docker version, just for troubleshooting, in case you have any issues, if you're tweaking things yourself, and then I set up a series of variables to build out an image name, starting with the learning-registry port 5000, layering on the repository name mongo-with-data, and then I'll tag this with ci- and then put the build number on the end. So as we run this over and over and over again, we can get a new image tag. Down below then, I'm performing a build to create this image, passing along that name that I just explained, and then I'm dumping out the registry, the learning-registry, catalog and the repository that we're working with. I'm dumping the tags for that repository and the entire catalog, just so you can see before and then dump the same information after, so we can do a before/after comparison by looking at the logs in Jenkins. And then I'm also doing a docker push right between these so we can push out that new image that we create to our learning-registry, which is that last piece of our process. So now we've got this job here that will fully automate everything after it gets triggered by a notification from the registry. It will fully automate building and then pushing the image to our other registry. So let's hop back over to the front tab of Jenkins here where we will see the result as things run. And now, let's go ahead and trigger this process. Now you could manually kick things off, but I'd rather let the notification from our registry do that. So let's go out and push an image to our vetted-registry and see if everything happens to then build and push an image to the learning-registry.

Simulating an Image Vetting Process

So I'm going to start out here with the very first part of this process by pulling down an image of mongo and then pushing it to our vetted-registry to simulate that vetting process. To better understand what happens as we push an image, on the very top panel here of my window, I'll go ahead and attach to the logs for the vetted-registry and pipe that through jq. And then in the middle, I'm also going to look at the logs for the learning-registry. I don't have those JSON formatted, so I'll just pipe those out without any processing. And then way down at the bottom here, I'm on my host and let's just say that I'm part of the security team right now and maybe I'm subscribed to a list of notifications about vulnerabilities in software that we use as a company. I get this notice about an update to mongo, so I want to pull it down to my own machine, so I do a docker image pull here, grab the latest mongo image, once that's pulled down then I have it locally, I can run all of my tests on it, maybe some tools that I want to run on it. And now let's say I'm done, I know that everything is fine with this particular mongo image, I want to push it out to our vetted-registry so that the rest of the company can just pull from that registry knowing that it's been checked at least by the security team. Do you remember what I have to do to push this out to the vetted-registry? Well first up, I need to name the image, so docker image tag here, take that new mongo image that I pulled down, and type in localhost:55000 on the start of the name, which is a registry, and then /mongo, and you'll notice this matches one of the repositories that I listed inside of Jenkins. You need the exact name of what you type in when you're pushing the image to show up in the list here. So back over at the command line, docker image push here, push out that new vetted image, and when I do this, I should start seeing output in the vetted-registry up above, including some notifications that are being sent. And if you take a look at the output of the vetted registry, you'll have a little bit of a problem. I left a challenge for you to test your understanding of what we've covered so far. So take a look at that error and see if you can figure it out and join me in the next video where we can troubleshoot what's going wrong.

Setting the Notification URL to the Jenkins Docker Registry Webhook

So if you take a look at RequestBin, everything looks okay here as far as the event that was received from the notification. But over in Jenkins land, nothing's happening. The trigger didn't go off. Do you know why? Well there's one thing we didn't do, and the logs for the vetted- registry point out what that problem is. You can see this error about sending to our Jenkins endpoint. What's the problem here? Well the problem is, if you look here, this is probably not the location of Jenkins. We did not configure our registry to point at our new Jenkins instance. So where do I go to do this? So I need to hop over and open up the config file for our registry. And I need to come down to the notifications section and I need to point this at Jenkins. Now another quick quiz, if you'd like a little bit of a challenge, what's the very first part of this URL to get to Jenkins? At least through the port that we need to use to touch Jenkins? Well we're using docker-compose, so we just need to use the name of the service, so learning-jenkins, and then we need the port that Jenkins is listening on inside of its container, which is 8080, and if you want to confirm that, you can look at the docker-compose file and you can see 8080 here for the learning-jenkins service. So that's the start of this and then I don't really expect you to know the rest of this unless you have looked at the Docker Hub notifications plug-in to Jenkins, or perhaps you peeked at the configuration file in the master branch and you saw this already. So the rest of this URL is right here. This is the endpoint in Jenkins that we'll want to send our trigger notification to. So I'll paste that in and make sure I save that file.

Image Push Triggers Jenkins to Build and Push Dependent Images

Alright over at the console, I'll just wipe out and recreate our vetted-registry just to reset everything that will, of course, include that config file change. And then I'll reconnect the logs to the vetted-registry only, since that's the piece I recreated. Now do you remember how I can double check my endpoint configuration for notifications? Remember up above here we should get some output that describes the endpoints we configured, and we can check here that our new URL to the Jenkins Docker registry webhook is set up correctly. It looks like it's good to go. And I can just pop down to the bottom tab here and I will push out that image, to try to kick off this whole process again. I'll speed things up here. Okay that is done now. If you look up above, we don't have some sort of repeating failure for Jenkins anymore, so let's look at Jenkins and let's just see what happened here. Hey, hey look at that, we actually have a job running right now, our pipeline is executing, and if I drill into it, you can see we've run docker version, if you'd like to see the output of that. And it looks like our build kicked off, you can take a look at the catalog for the learning- registry before and you'll see there are no repositories. I'll zoom in a little bit here for you. You can look at our particular repository and, of course, that doesn't exist yet. And then I push out the new learning-registry port 5000 mongo-with-data:ci-1 tag, it looks like it's pushed out. This is pretty darn cool. And then if I curl again to take a look at that learning registry catalog, you'll see we have our repository and down below, if I look again here at that repository, take a look at the tags for it, look at that, we've got ci-1. Is that not cool or what? Just for fun, let's go back a screen and let's run this build again. So let's push another image. So this is a status page, open this up so you can see the status on a high level, and then let's go over to the command line, down to the bottom tab and push again, I can kick this whole process off. Of course, it's the same image, so not much will have changed when that build runs, so it should run pretty quickly here. Here's run number two, and yeah, it's done before I can even drill in. And then neat thing is, if I look at the last step here, ci-2 is now included and I can just keep doing this over and over and over again, kicking off as many jobs as I want to build out new images that are then automatically pushed to our learning-registry. And of course, if nothing has changed, all it's going to do pretty much is retag the image that we already have. That's a beautiful thing about Docker layering. If we zoom out here and look at our overview, you can see we now have a whole process set up to automate one team vetting images and pushing them to a vetted registry, that vetted registry then tells another team, specifically it tells their Jenkins instance to go ahead and kick off a process of automating the process of creating their images that are built on top of security vetted images that are then pushed out to a learning registry so students can grab them. This is all a really neat demonstration of the power of registries, of Docker, of even Jenkins here to spin up a whole environment to do this, even if just for learning purposes, and start to see all the benefits of registries at play. And that brings us to the end of this module. If you want to tear everything down that we just set up, use the recreate file, just run that very first docker-compose down command with the additional flags to tear everything apart and get rid of everything that we created, aside from the images that were created, you can clean those up by wiping them out manually or add a flag to docker-compose down to do that for you. The rmi flag will help out. And as we're winding down, I would really encourage you to start to think about the possible use cases of reacting to people putting images inside of a registry and what you might want to trigger to happen as a result of that and if these notifications that can be embedded inside of your registry would be beneficial for those triggers. And then as you're thinking of that, let's turn our attention to the next topic, which has to do with cleaning up images inside of our registry and talking about the configuration of storage.

Cleaning up Images and Configuring Storage

Sadly You Cant --force-recreate in Production to Free up Space

I'll admit it might be hard to visualize a reason for wanting to clean up images and perhaps even to change the storage of images, given that in all the demos in this course we've just been recreating everything when we need to wipe things out. But the reality is, you can't do that in your real production environments. You might not even be able to do that in some of your testing environments because you probably want to keep your image history. At the same time, you can't just have images building up forever. Otherwise, your disks are going to fill up and you're going to come in in the morning and you're going to find that your server has crashed because there's no disk space left. Yes, I've been in that situation, pulling my hair out, trying to figure out what exactly went wrong because nothing should have gone wrong. And then all the sudden, I realize that the images are a lot bigger than I thought and I've been piling these up from all of my testing, and then, of course, I have to recreate servers if that's easy to do, and in my particular case it was easy to do, but that's not always the case, so we don't want to have to do that. You don't want to have to go in and troubleshoot and manually free up disk space and go into maybe a safe mode to do that. So let's actually talk about how we can make sure that images don't just pile up and fill up our disk.

Registry Storage Drivers

So one approach we have to clean up is to blow away the volumes that we're working with, and in the last module I showed you how to use this new docker-compose flag, added in 1. 19 of docker-compose, this renew-anon-volumes flag, or a -V for short. And this went ahead and wiped out the anonymous volumes and gave us new anonymous volumes so that when we restart our registry, we wipe out all of our images, and that works great for us when we're testing here. But there's actually an easier way if we're just testing the registry and we don't want images to persist very long and we'd like to just be able to restart the registry to wipe them out, there's an easier way to go about this then, we can instead configure in memory storage of our data. So data being whatever it is that we need to store images that we're pushing and pulling. So let's take a look at this in-memory option and this is available thanks to the pluggable nature of the registry, specifically there are storage drivers that we can work with to configure how exactly we store image data. And while we're on the subject of the pluggable nature of the registry, one thing I highly recommend is to go take a look at the source code for this and take a look at the interface for a storage driver. So if you hop out to the Docker distribution repository, out on GitHub, and if you look then inside of this storagedriver. go file, you'll see the contract that a StorageDriver implements. You can even clone this down to your computer and browse around inside of Visual Studio Code or whatever IDE you use and learn a lot more about how this works. So for example, you can see methods in here to get content and to put content into whatever storage system we're using. And further down, you'll see various commands to operate on that content, to list content, to move it, to delete it. So there are a lot of different operations in here, all of which define a storage driver. And if you're ever curious what a storage driver is capable of, well then go look at this interface because this is going to describe the behavior of a given storage subsystem. Anyways, one of the implementations of this storage driver is an in-memory storage driver. And if you're curious, you can look through the files here, and if you look at the results, for example, in Visual Studio Code, you can see a list of a couple of matches when I type in the word memory, and the ones you want to look for are under the distribution/registry/storage and then driver folder. And here is the in-memory driver. And if you peruse through here you'll see the implementation of those methods we just discussed, for example, GetContent and PutContent. So I really encourage you to poke around in here and learn a little bit more about what's going on behind the scenes, I find that this really helps demystify things for me. Especially when I think the documentation might be a little bit lacking. Take advantage of open source code. And then in addition to looking at that one in-memory storage driver, if you pop open the file system and look at that entire driver folder that we were just talking about in the path, you'll see a series of other options, for example, Azure or the filesystem driver that is the default. Also there are some choices here for s3 and swift and you're going to find a host of different drivers that are already available to you that you can just plug in via some configuration changes. And speaking of that, let's talk about setting up the in-memory storage driver. Let's do that next.

Switching to the in-memory Storage Driver

Alright so if I want to use the in-memory storage driver, what do I need to do to set that up? That's something we need to configure inside of the registry, so we need to go to the config file for a registry, or use environment variables, and we need to turn on this other storage driver. And I've already done that, I've set up an example in the course repo. If you go into the storage folder and then inside of the inmemory folder, I've got a docker-compose file set up here with everything you need so that you don't have to type this all out or watch me type it all out. So I've got a registry set up, mapped to port 5000 on my host, and then I've configured the inmemory storage driver, and there are no options to this, so you can just simply specify that string there after REGISTRY\_STORAGE. You could also use a config file for this, but I figured hey this is just one feature we need to toggle, why don't we use an environment variable for that. And then down below I've got some commands that you can run that I'll run here to test this out and see the effect. So, let's do that, let's hop over to the terminal, inside of that storage inmemory folder, and you'll see the compose file there. And let's go ahead and bring up this registry. I'll leave myself attached so we can see the log output here. And you're really not going to see anything too exciting in the output, at least not right away, that indicates what storage subsystem we're working with. If you are curious about that, one thing I've found that can be helpful is hop over to the config file here and add in another environment variable to set the log level to debug. This will print out a little bit extra information, if I save that. Back over at the command line, I'll kill things here and bring things back up, and now if I wait a second here, you should see some log output, there you go, we've got some debug output. You can see the level=debug here, that's what we get by turning that level up a little bit, we have a little bit more diagnostic information and you can see we have some inmemory tracing going on, in this case, a trace of the stat operation. So that confirms that we are running inmemory. Next up, let's go ahead and test this out.

Restarting Wipes Data with the in-memory Storage Driver

Alright I split the screen here, I've got my log output up above. Keep in mind that's going to be a bit chatty, which is fine. Down below I'd like to monitor what's going on with the registry, I'd like to monitor the list of repositories as we make changes. I want to do this to confirm that when we restart the registry we do indeed wipe everything out because we're working inmemory for storage. So my question to you now, how can I check the list of repositories in this registry? Well how about we just query the registry API. And do you remember how to do that? How do I get a list of repositories from the API? Well I need to access the v2\_catalog endpoint. And to do that, I've got a watch command set up here that will pull this every second and look for the list of repositories, that way we can watch the list as we make changes. So let's go ahead and run this watch command. And if you're not familiar with the watch command, it just takes a command, like curl here, and will run it repeatedly at whatever interval you specify, in this case you can see every 1 second. And over in the upper right, you can see the last time it was run. Okay so here we've got our output every second, this will refresh on the bottom and when changes happen the changes will flash at us, that's what the -d flag to watch does, it highlights differences. Okay so now I'm going to split the screen yet again here so that down below we can push and pull images. And I'm going to work with BusyBox, it's a lightweight image, easy to push and pull quickly. So I've pulled that down. So I'll rename the image, give it another tag or a name that points it at our registry. Remember I'm on port 5000 now for this inmemory registry. And then I'll go ahead and push this. Keep an eye on the two panels above, notably the watch command. So I'll push this out to our registry now, and you can see right in the middle we now have a repository. So now people could push and pull from that repository to get access to this busybox image. And of course, the big test is if we restart the registry, what happens to that repository? So let's do that next. And can you tell me how to do that? How can I restart the registry? Well we're working with docker-compose, so we can just do that followed by restart, then wait a second here. Watch the difference there in the middle and, there you go, you can see our list of repositories is wiped out. So we are indeed working with an inmemory storage subsystem that just wipes everything out when we restart at any point in time. And that could be because of a failure. So use this sparingly when you're testing out working with the registry and you want to learn what's going on, this is a great use case for the inmemory driver, it's not a good use case though for a production environment. It's really not even a good use case even if you want a volatile storage subsystem in some sort of production environment, say you maybe have a mirror and you don't care if you lose the contents of that mirror from time to time. If you would like something similar, what I would recommend is going back to the filesystem driver and map it to a ramdisk instead. And speaking of the filesystem driver, by virtue of the fact that you could be mapping that to any sort of volume or bind mount onto the host, means that the underlying storage technology that your filesystem is provided by can give you another degree of flexibility. So I want you to think about that too. Think about the container volumes at play and how that might give you the flexibility that you want when it comes to storage in general and maybe even the consideration of how things are cleaned up if you want something to be wiped out. So if you already have some other storage system externally that you're mapping through to volumes with the rest of your containers, well that might be something you consider and not even necessarily look at a pluggable storage driver inside of the registry. And now the converse is true as well, if you don't want to lose your registry data, if you have important images that you're keeping, considering the container volume that you're working with or bind mount or whatever it might be that you plug in for the storage driver, you need to think about how reliable that storage subsystem is and what could go wrong with it. For example, you might have availability issues or scaling issues or just data integrity or data loss issues. You need to think about all of that if your registry data cannot be lost at any cost.

Cloud Specific Storage Drivers

While we are on the subject of storage drivers, I would strongly encourage in your checklist, when you move to a production environment, to take a look at some of the cloud specific or stack specific drivers that already exist. We just saw some of these in the source code for the registry. There's also another good spot to go to learn a little bit more if you're not up to perusing through source code to learn what's available and you'd like a high level synopsis, well hop out to the docs for the registry configuration and then out in the docs over on the right hand side, you'll see a section for storage, click on that, and you'll get a list then of all the options for all the different drivers that come out of the box, which is great because this provides a list of the drivers you might want to consider. Here's filesystem, here's Azure, here's Google cloud storage, here's S3, and with S3 keep in mind that other systems might provide an S3 compliant API and therefore work with this driver, so that'll be another choice, and then Swift and OSS, and if we keep going you'll see other options, but the nice thing is we have a list here of the various different cloud platforms and then the configuration options within each. That starts to give you an idea of what you might want to take advantage, depending on where you're going to deploy your registry. If none of this works, then you can always write your own custom driver. I would discourage it, I would encourage you to try to find something that somebody else is supporting and really think about your use case before you do this, but hey the sky's the limit and when you have open source code with an interface that you can implement, you can always customize things exactly to your liking.

Deleting Images Example Setup

Now while we're on this configuration page looking at options, one that might stick out when we're talking about the subject of cleanup is this delete option. So obviously wiping everything out is one route to go, but it doesn't work in every situation. If you have data that you would like to narrowly target to remove, so a set of images that you'd like to just get rid of for whatever reason, well you'll need some ability then to target those for deletion. And it just happens to be that by default deleting is turned off. So let's take a look at setting this up and let's take a look at using it. And for this particular example of deleting images, I've got another example setup in the repository. I marked the starting point with delete-starting-point in the branches out on the GitHub repository so you can pull and check out from there if you'd like to start at the same point that I'm at. And then go into this storage/delete folder inside of which has a few files. We have a compose file, we have two folders for the two services that we'll be working with, one is the registry, and then the other one is a front end for the registry, so we have something to go click on to delete images. Otherwise we're stuck using the API and there's nothing wrong with that, but hey, it doesn't really match the workflow we might like in reality to be able to go one off delete an image, and to really get that idea or workflow of sending deletes into our mind, I've gone ahead and set up a UI component here that can do that for us. And it just so happens that that UI component has configuration and that's what this config. yml file is for. So it uses yml as well. Now the registry itself, I have a Dockerfile and then a configuration file. Let's hop over and I'll show you the Dockerfile. All I'm doing is adding the tree command in because we're going to go into the file system for the registry and I'm going to show you the storage of image data on the filesystem and the tree command is great to show a hierarchical view of that. And then as far as configuring the registry goes, nothing special here other than the defaults plus JSON output in the log. And then as far as the front end component is concerned, it's just a standard default config. I actually copied it out of the container image and then I added a few lines to explain how I did that, otherwise these are the defaults, and then I also set a URL back to the registry that I spin up here so it can talk to that registry and show images and eventually delete images. So we've got that config file, we'll close that up. And let's take a look at the compose file. So inside of here, we just have our two services, registry and the front end or web. In this case, I'm using a different image because this one actually supports deleting images. The other one we used earlier in the course, it had the functionality in v1, but not in the v2 support for the registry, so I've abandoned that one just for this demonstration purposes because I thought hey this would be a great opportunity to show you that there's more than one front end you can put on a registry. And when it comes to what you might want to choose in a front end, you might want to consider if it has delete support if that's something you're going to use. It's not a necessity because it's very possible you might just be executing delete via an API, in which case you don't need the front end. So anyways, that's this component down here. I have opened up port 5001 to that, I have port 5000 to the registry if we want to look at it, and then I've mapped in volumes for both of the config files that we just looked at. I do have some commands down at the bottom and some links if you'd like to learn a little bit more or just copy and paste some of the commands that I'm using in this particular demo and some subsequent demos. With that, let's start this up. So over at the command line, all we have to do is a docker-compose up and we are good to go. That will build the custom registry image just layering in one command for us and then it'll boot everything up. Once everything has started up, and that front end can take a little bit of time, you should be able to come over to a browser, open up port 5001 on your host, that's what we publish for the front end, and you should get this list of repositories and it should just be empty at this point in time. Okay so we've got everything spun up.

Exploring the Registry Filesystem Layout to Store Images

The first thing I'd like to do is come over to the command line and toggle over to a new tab, we'll leave the logs in the first tab, and then in a new tab, I'd like to get into the registry itself and look at the file system. And keep in mind, because I reverted back to the default configuration for the registry, we're now using the filesystem driver again underneath the storage node here, so we're back to storing images and you can see here's the path that we can go look on disk. So why don't we take that and copy that and let's hop over and get inside of the registry container and look around. Now I've got a quiz for you, do you know how I can get in there and take a peek around the file system? Well that sounds like a great opportunity for a docker-compose exec. So I'll run that, specify the registry service, and then I'll use just sh for a shell. Once I'm in there, I can peek around. And why don't we change into the directory that we saw inside of our configuration file, so the default config says, hey go ahead and put the stuff here. And if you remember earlier in the course, this is the location where we created a named volume so that we could have a longer lived persistence of images on disk for our registry. So we are in here, if we take a look around, we have nothing. That's not too exciting. Why do we have nothing here? Well we have nothing here because we haven't put anything into our registry to have. And until we do that, we won't get any of the structures that we can poke around inside of. So let's do that. While we're about to do that though, I would like to just watch the listing of files or the directory structure here so that when we push something, we can see it all materialize. And to do that, I'll use the watch command every second. I'm just going to run a tree command here, and I'll split the screen first so I can push my image down below, and then I will run the watch command up above, so we can see simultaneously the changes as we push an image. So watch is running up above, you can see no files, no directories inside of the stored location for our registry. And then if I come down below here, I can push an image. And how about we start with a busybox image. So I'll push that up and, bam, take a look at that on top. We have this huge structure that comes to life. The first thing I'm going to do is just detach from watch and clear up above and just start navigating in a little bit here because we have a rather flat structure, at least a few levels deep where we're just going inside of a folder, inside of a folder, inside of a folder. So I'll come in here and now we'll look around. We've got two primary folders. We have blobs and repositories. And if we take a look at the tree command here, not as daunting because we got rid of a few layers on top there. You can see we have three separate blobs stored. Blobs are basically just data files. For example, a data file with an image layer or an image configuration or an image manifest. So the actual data that comprises an image, that data is stored inside of blobs. And then to organize those blobs, a content digest is created, which is a hash of the contents of the data that is stored in a blob, and then that is used to build a folder hierarchy to look up a given blob based on its content digest, so it's an index by digest. And then down below is the repositories folder. You can think of this as a second level index for us to look up blobs. For example, we have the busybox folder here under repositories. Underneath that then we can go into manifests, we can find the latest tag, and we can use that to find the content digest of the blob that contains the manifest for the busybox image. And if I come up, you'll see the same digest up here, which means the data file here has the image manifest that we're looking for. Let's open a new tab and take a look at that data file. Now I've already connected up with docker exec and I've navigated into the content digest that we are looking at. So let's cat out the contents of the data file. And there you go, we have an image manifest. If you're not familiar with what a manifest is, well it basically describes the layers of an image, as well as where the image configuration file is at. So it's all the metadata that points at the various pieces of an image. Now I want to challenge you to pause the recording and stare at this manifest and see if you can find something interesting that links back to the file system organization we were just discussing. So I was hoping you would notice the content digest here, the one that starts with 8ac for the image configuration and the one that starts with f70 for the single layer in this busybox image. So when you Docker pull busybox latest, this manifest is retrieved, these digests are extracted, and then the digests are used to look up the corresponding blobs inside of the registry. As an aside, I chose the busybox image because it has a single layer, most images will have multiple layers so you'll have multiple blobs then, one for each layer. So don't be surprised if you're looking at a different image if you have quite a few more digests to look up I like busybox because it's easier, there are fewer moving parts to track down. Of course we can continue down the rabbit hole here. The image configuration is another file that we can easily look at, another JSON file. So I'll open that in a new tab for you. Before I switch tabs, take note of the 8ac at the start of the digest for the image config. Here in a new tab, take note that I'm inside of the 8ac folder and I'm dumping out the data file, which contains the image configuration and this is JSON, it is not nicely formatted though so it's a bit hard to read through. Nonetheless, if you look through here and if you're familiar with image configurations, you'll see things like the entry point. Here's a working directory and I can't quite tell if this was the working directory at the time the image was built or the one that will be used when you start it up. This is a bit of a mess. I might like to pipe this through jq if I'd like to learn a bit more, you can do that on your own, but I hope you're starting to understand that when we store on the file system or the filesystem driver, we've just got all the blobs that are a part of our images, which might be a manifest or a layer or an image configuration file. We've got all of those stored on disk and we have references to those. Now it's a great organizational structure, it's content addressable storage, the digest is based on the content of these various blobs. Consequently we can deduplicate our data. For example, if two images share a common set of base layers, we don't need to store the blobs for those base layers multiple times. Each image can just reference the same set of blobs. That does make things a little bit complicated when it comes time to delete because we need to make sure that there are no references to something before we get rid of it. Otherwise, when we delete one image, we might be deleting layers that are actually used by another image that we still care about. So that's something we have to think about. If the structure of an image has peeked your curiosity and you would like to learn more, for example, about image manifests, image configuration, and even layers of an image, check out my course Containers and Images: The Big Picture. Notably, I have a whole module on what exactly is an image? For now, let's turn our attention back to deletes, and I would encourage you to keep an eye on the file system as we perform deletes to see what changes behind the scenes. That'll better help you understand what all is involved when it comes time to clean up images.

Enabling Deletes in the Front End

Okay so here is our web UI and there's the busybox image once I refreshed. And inside of here, we have the latest tag and you're probably wondering how in the world do we delete things, there's no delete button here. Quite a few of the front ends that you might encounter may have some sort of toggle that turns on or off the ability to delete images or perform other administrative tasks, and so you'll want to keep an eye out for that, and in this particular case, with this particular front end, there is a toggle that we need to turn on. Do you want to guess where that toggle is at? It's inside of the registry front end configuration file, which is why I extracted that for you in the starting point for this exercise. There's a read only element in this case. Now keep in mind, this is the front end, this is not the registry itself. So I just need to set this to false to be able to delete images from the front end and save that. And then I am going to need to restart the front end. For that I might open a new tab, not to disturb any of the rest of what we've got going on here, and I'll do a simple docker-compose and a restart here, and I can narrowly target just the front end with the web service. So I'll leave the registry up because I didn't make any changes to it. Now while that's restarting, I would like to come over to the tab here where we ran the tree command and copy that information out because one helpful way to learn what's going on is to open up a dif tool and to dif the changes to the file system before and after we perform our delete. So right now on the left, and I know this is small, but I'll explain it and we'll zoom in if we need to, right now on the left we have the structure of the file system, after pushing the busybox image, but before performing a delete. So now let's go do the delete. So let's hop over to the front end and refresh here, and we now have a Delete button, and that's great. So we can click on that and we'll get a little prompt that says are you sure you want to do this? And of course, we're sure we want to do this, and we click the Delete button and we should get another problem. And I left this one as well, I could have turned it on by default before I set the starting point, but I really wanted to stress that the registry itself has another toggle that needs to be turned on if you want to support deletes. So we've got this error right here, if you want you can click to get more information, and that's a really helpful link because it jumps you right into the registry configuration, right under the delete configuration element and we can really just pick this up and copy it and move it over to our registry configuration. So where do I put this? Well this is registry config, so let's hop over here and go into our config. yml file for the registry, inside that registry folder, that's why I split these out, a little bit easier to see the structure. And then come right under storage here and just drop in delete and then make sure enable is nested underneath. So we need to enable deleting. Okay so let's hop over to the command line and I'll just kill off the docker-compose up to detach from the registry and the front end. And I'll just run docker-compose up again so that I restart and, of course, attach to the logs if I want to see those. Because I restarted everything, I need to hop over and exec back in to the new registry container. That way we can take a look at the file system again.

Digging into What Happens When You Delete an Image

So let's first CD into the registry data directory. And first up, let's dump our file system and copy that and bring that over and paste that in and look for changes. There is one inconsequential change that I don't want to discuss, so I will paste this output as the starting point on the left hand side for any of the difs we're going to do subsequently. So just ignore that new a3 blob. Alright, so let's do our delete now and see what gets removed. So I'm back over in the front end and I'll click delete again, and hopefully it doesn't error out yet again. Alright, and it looks like everything is gone now, so you might be tempted to think that the file system is entirely cleaned up inside of the registry, and then you could go about your day and not worry about this anymore. But we need to take a look behind the scenes at the file system to see what actually happened. Let's clear this out and do our tree again and copy all of this and come back over to our dif tool and paste that on the right hand side. So this is the result of our delete. The gist of it though, if you look down below here, we've pretty much just removed the tag for the manifest and that's really it. We haven't deleted anything else on disk, so if you look up above, our blobs are not touched. That means the manifest for our image, which is this 1866 here, as well as the f70 blob for the layer and then our image config under 8ac. So, if you will, we've deleted the pointer to the blobs that are a part of our image, but we haven't actually got rid of anything that matters when it comes to appreciable size. So just to recap what this looks like, we have our latest tag that we're familiar with working with for our busybox image. That is pointing at the digest, which started with 1866, that is the digest of the manifest of our busybox image, and we printed that out and we saw inside of there, there were a couple of different links or references, one of which was to the image configuration, which started with 8ac for its digest as well. Remember this is all content addressable. And that was the JSON configuration for our image so that when we spin up a container based on it, we know the application to run, for example, or the environment variables to have by default. So our default environment is inside of the image configuration. And then we also pointed out the layers for our image and that would be the f70 prefix, and in our case, thankfully, with the busybox image there's only 1 layer, so it's not like 10 of these to try and figure out and track down. But there could be more because we could have a stack of layers involved in an image. And so we have these references and all we did in deleting things was to basically get rid of these pointers over on the left hand side. So we've got rid of the latest reference, we're not going to see our image then when we query via the API, nor our repository anymore, which makes it looks like to us humans that everything's cleaned up, but it's not. And that's okay. We have basically a two phase process here and we've done the first phase and that's just deleting these references that we're used to working with to the manifest for our image. And in fact, if you look at things behind the scenes, you'll notice that we specifically deleted this particular manifest digest, the 1866 fingerprint, when we clicked that Delete button. And if you look in the log, you can see the delete take place against the manifest 1866.

Deleting from curl or a CI Tool Like TeamCity or Jenkins

One last thing about deletes, you don't have to use the front end, you're perfectly welcome to go straight to the API itself and I've put in a curl call that will do that inside the docker-compose file for this delete example. If you'd like to try that out, you might need to change the manifest digest, the last part here after the sha256, to match whatever it is that you have pushed to your registry. So figure out the digest for your manifest for the image you pushed out and put that in, and then you could call a delete here from curl from the command line, instead of from the web UI. And that's really important because there are a lot of tools, like a ci tool, that can keep track of the images that you create as a part of building your application, if maybe you're using images to distribute your application. You ci tool can keep track of these identifiers for the digest and when you tell your ci tool, like maybe Jenkins or I know for sure TeamCity has this, when you tell it to clean up a particular set of builds of your application, it can actually go out then and use these digest references to go clean up your registry for you. And of course, that's a pretty big deal to clean up because images can pile up and take a lot of space, especially if ci is building a new image every time somebody checks in code.

Garbage Collection Dry Run

Now let's talk about the second phase of deleting, getting rid of the actual data. And I believe, seeing one more thing of the blobs involved in the busybox image will help you really understand and drive home the point that the disk usage is in these last bits we need to get rid of. And that thing I want to show you is that f70 layer. Let's take a look at that blob. So I've already opened a new tab here and I've drilled in to the storage location for that f70 blob. I'm sending in the bucket of all things that are a digest starting with f7. And inside of here just right now is that f70 folder then. Inside of there is a data file. So let's go ahead and take a look at what this file contains, and to do that I'll use the file command. Which, by the way, I cheated. This is not installed with the registry image by default, I had to do an apk add and then file, like with tree, you can see that in the Docker file for this example if you want to add that in. Anyways I added the file command so I could take a look at this file type, and you can see this is a gzip compressed data file. So if you've ever worked with archives, what could I do here to take a look at what's inside of this archive? I can use the tar command and I can test or list the contents inside of this archive and I can point right at that data file then. What does this look like to you? We've got the contents of what looks like, and there's a lot in here, what looks like a file system, and so as you can imagine, this is what's going to take up the space. And of course, the busybox image itself is not very big, nonetheless, you could image that this could be pretty large, for example, if we're working with the mongo image or any other image that's a couple hundred megabytes instead of maybe only a couple of a megabytes. Now that we understand what's inside of that f70 layer, let's go back to talking about the second phase of deleting. And as a very first step, I'm going to take the test on the right and copy it over to the left so we can only see changes after we perform this second phase of deleting. And this second phase of deleting is referred to as garbage collection. Just like with object oriented programming, if we don't clean up unreferenced objects, we could have memory leaks in our application. Same thing here, we need to clean up unused blobs to reclaim disk space. So the registry comes with garbage collection code. If you come over to the terminal inside of the registry container, you'll find a registry binary and you can grab some help for it. When you do that, you'll see that the binary that's running our registry has actually a series of commands that are available. We have the serve command, which we're using to serve up images and to provide the API that we interact with, but we also have the garbage-collect command. And if we'd like to know a little bit more about that garbage-collect command, we can put that in here and we can get some help about that. And fortunately when it comes to garbage collection, there's a dry-run flag that we can provide that will tell us what exactly garbage collection is going to clean up before it actually gets rid of it. And that's a good thing. So how about we do, how about we just take the registry binary, we'll perform garbage collection, but we'll do it without actually applying that garbage collection. This takes one last parameter. We need to point at the configuration of our registry so it knows where to look to clean things up. Where do you think I can go to find that at? Where can I get the path to our registry configuration? Okay so the answer would be, let's look at the compose file. And we actually have this path, right where we've mapped in our volume that contains the configuration that we're overriding. So I'll grab that path in the container and come over and paste that in the terminal. It looks like that's wrapping a little bit weird, that's okay, we'll go ahead and run this now, and you can see this returns back a series of blobs that are eligible for deletion. We have four of them. And if you look at the identifiers here, well actually it's 4 and we only have 4, if we go back to our dif here, we've got 18, 8ac, we've got a3 and f70, so those are the excate 4 that are involved with our busybox image. Now obviously if there are other images out here or different tags that have different manifests behind them, different digests, then there might be some other blobs that are not marked. So what's going to happen is if we run garbage collection, these will be removed.

Safely Running Garbage Collection with Readonly

So let's go ahead and run our garbage collection. Before I do that though, I would like to point out one important thing that is a consideration you should make, and that's whether or not you're going to be using your registry when you perform garbage collection. It's actually recommended that when you go about garbage collecting these various blobs that are no longer referenced, it's recommended that you put the registry into a read only mode, otherwise what could happen is somebody could be pushing an image and part of the layers for that image could be pushed up whereas the rest of it's not quite pushed up yet, and it's possible that part of a push that's happening right now when you run garbage collection could result in some of the layers that are a part of that image being deleted. And if that happens, then you could have a corrupted image stored in your registry. So it's a good idea not to be writing to your registry when you're running garbage collection. This is referred to as a stop the world garbage collection. We need to stop everybody else from doing anything. So this might be something that you do at night or on the weekends when nobody is using the registry. Of course the safest bet is the read only mode. Do you want to take a guess where we go to set that up? Well that's going to be in the config file, so I'll pop over to that for our registry, and then inside of here, right below delete, I'll add maintenance section. Nested inside of that, I'll add readonly and nested inside of that there's an enabled flag. So I can set that to true. And if you're curious, you can refer to the docs. On the right hand side, you'll see readonly under storage, you can click on that, and this is actually a little bit misleading here in the table of contents because this readonly flag is set underneath maintenance. So there's maintenance and nested inside is uploadpurging and readonly. It's a little bit easier to understand how to set this if you come up to the overall storage configuration example and scroll through here, you'll see maintenance and then nested inside is readonly, under which you will set enabled to true to enable readonly mode. So this is one of two maintenance features, there's also uploadpurging. By default uploadpurging is turned on. This is another aspect of clean up. I won't run a demo, but I want you to know about it. The gist is, when images are pushed or uploaded, if there is any orphaned files as a part of that, which can take up disk space, those are cleaned up during uploadpurging. Anyways, we're going to look at the readonly mode, we have set that inside of our configuration file and I've saved that config file. Now what do I need to do? Well I need to restart the registry to pick up this new maintenance readonly mode. Unfortunately I'll get kicked out of my exec into the container, that's okay, I'll go back in. For now, over on the first tab, I will kill off everything, Control+C to docker-compose-up, the call from earlier on. If I do that, I stop both of the containers. Then I'll clear out and run docker-compose up again to start things back up. Note, I'm not force recreating anything, otherwise I'm going to lose what I have stored on disk with my registry, if I wipe out my volumes with that -V flag. So we've essentially performed a restart of the same containers. Now that things are back up and running, let's go over to the exec tab and let's re-exec in so we can run garbage collection.

Running Garbage Collection

To save some time, I've exec'd in and I've pulled up the command that we're going to run, this time without the dry run flag. And now when I run the garbage collection, you'll see that 4 blobs were eligible for deletion that are a part of the busybox image, that's up here as well. And you can see in the messages here, we deleted each of these blobs then. So let's clear this out and let's take a look at our file system again. That looks a lot better. So let's copy this and let's put that into our dif on the right hand side here, and compare what happens once we perform the garbage collection. So now you can see the blobs are gone up on top, including the files that are taking up the most space. And so we've now truly freed up disk space. So it's a two phase approach, you need to decide what you want to delete and that could happen any number of ways via the API. I would strongly recommend that you look at automated approaches for marking things for deletion so that you're not going through and manually deciding what to get rid of, have some policies in place for that. And then also I would recommend, if disk space is low then run garbage collection. And maybe before that point I would really strongly recommend that you just have a lot of space available for your registry. Space is cheap, there's no reason to be cheap with the amount of space that you give to a registry. And then you can run garbage collection when it's convenient, maybe just on the weekends or sometime when truly nobody's using your application. It's also worth stating that you don't have to put the registry into readonly mode. If that's too much of a hassle and you know that it's safe to leave it operational for writes, then go ahead and just perform your garbage collection.

Tag Mutability and Garbage Collection

While we're on the subject of cleaning up, there is a feature that you'll want to watch out for in an upcoming version of the registry. It's very likely you'll be interested in this, so let's understand why this flag is being added. And to do that, we need to talk about tagging conventions. One possible model for tagging is to use immutable tags that never change. So for example, in your ci environment, each build of your application will have its own unique tag and that tag will never be reassigned to another build of your application. The fourth build in ci will use ci-4, you will not reuse ci-3, or maybe you use an incrementing version number for your application. This model jives well with everything we talked about in this module. For example, if we're done with ci-1, we can use the tag to look up and delete the reference to the manifest for that image, and then later on garbage collection will wipe out the blobs. So this is an immutable tag model, you're not changing the image that a tag points to. It's not the only model though because tags are references and we can modify those references, for example, in a different scheme you're probably familiar with. If every time we build our application in ci we rebuild the latest tag and push that out to our registry, then we're going to leave behind a history of images that are no longer referenced by a tag. This is not at all problematic, it just means that you can't use your latest tag to go clean up the previous latest images. You have to delete those images by the digest of the manifest, so that second level lookup is no longer there. And that may not be a big deal, for example, if your ci environment is keeping track of those manifest digests, then it doesn't need a tag lookup to clean things up. However, this can be a sticking point because a lot of people don't really care about referencing images by digest, so they assume that if you update a tag to point at a new image, the assumption is then made that the old manifest is dangling itself. But the reality is, we can pull images by the digest alone, we don't have to use just tags. So the registry can't make the assumption that an untagged manifest is dangling. You have to delete the manifest specifically. This purposed new feature would consider manifests that are untagged as dangling or in need of garbage collection. So if this is more your style to assume that an image is no longer needed when there's no longer a tag for it, then watch out for this new feature to land, it should be coming in a next version of the registry. So the proposal is to add this -m flag. With this new flag set, once a manifest is no longer tagged, it will be garbage collected as well. If you want to avoid this problem of untagged manifests, one thing you could consider is always tagging both with an immutable tag, so ci-1 on the first build of ci, and your latest tag or whatever your tag name is that gets updated. So tag the same image multiple times and as long as one of those tags is immutable, the ci-1 here, then you'll always be able to find that image by your tags. So the next build here will be ci-2, that's the immutable tag, and latest will just update to point at that image. And likewise with ci-3 and so on. The nice thing here, you have a rich history of individual tags that are immutable for every build of your application, as well as the latest pointer if somebody just wants the latest version of the app. The alternative to this, you could delete an image before you update your latest tag, so before pushing new images, delete the existing image if you're going to reuse a tag. By the way, if you'd like any more details about garbage collection, hop out to the docs and check out the specific article with regards to a registry and specifically with regards to garbage collection. And with that, we are at the end of this module, let's move on to the next module and talk about registry security.

Securing the Registry

Time to Secure the Registry

At some point after seeing the features of a registry and being convinced that the registry is valuable for a couple of different use cases at a minimum, at some point we have to step back and talk about some of the aspects that we need to consider, for example, to secure our registry. That way, when it's time to put a registry into production, we don't just have awesome features, we can also lock things down and make sure just the right people have access. And what do I mean by this? Well, right now we've got this HTTP API that's provided by the registry that we can use from a web browser here if we make web requests to it, or that the Docker daemon itself can interact with via the Docker CLI. We've got this API opened up and potentially exposed to the entire world because we have not locked anything down yet. Now a couple of times I've talked about how you can isolate that registry so that traffic coming from other computers on your network can't get to it, but at some point you might want to expose your registry to somebody else, besides just your local computer, and then you might want to control what they can do with it. So that's one of the aspects of securing a registry that we can talk about. So why don't we start there. Because right now, as you can see, I can make a request to our registry on port 5000, I can grab the catalog here, and I can see our list of repositories and I've got this busybox repository in here. This is running on my local computer, and if I hop out to my Windows machine, and this is also a separate machine on my network, if I hop out to this computer, I can reach back to my Mac where I'm running the registry at and I can access the exact same API. We even saw how we could use this to pull images to another machine, this is really valuable, but obviously we need to lock this down. So let's first talk about some forms of authentication. And the most basic form of authentication is basic authentication. So let's take a look at that first.

Basic Auth Limits Client Access with an htpasswd File

Now before we dive into the implementation of basic authentication, let's just step back and make sure we're on the same page. Now I want to do this by walking through a high level diagram because we're going to talk about several different security scenarios in this module and it's easy to get these mixed up. So let's get our bearings straight first and then let's dive into this basic auth example. So we've got our private registry deployed and we've been talking about the benefits of using this to access images on various different workstations, perhaps internal to our organization, perhaps just at home, and also to access this registry from servers, to deploy applications. Applications that maybe depend upon sensitive images. Now at some point in time, you're probably going to want to share these images with somebody outside of your organization or perhaps even on a network inside of your organization, a network that you just can't quite trust as much. So you're going to need to lock this thing down. That way, once you start to allow the outside world in to get to the images on your registry, you can have some sort of control over who can and who cannot have access to those images, especially if you open this up to the world wide web. Now I find it helpful to delineate two perspectives of the security discussion. The first perspective is making sure that only specific clients have access to the images in the registry, so it's kind of like the client-side of things. The flip side of that is the server-side of things, for example, making sure that when we talk to a registry that the server is actually who we think it is. And when we talk about basic authentication, we're talking about the first half of that equation. We're talking about a means of locking down the registry and then distributing keys to various different clients to give them access to that lockdown registry. So this is how we can ensure that only specific clients have access, the client-side of things. Don't get that confused with the server- side of things, of the server proving who it is. We'll talk about that later on. And of course, when it comes to ensuring only specific clients have access, there are different techniques we can use. Basic authentication is just one of those techniques that more or less stores the user names and passwords to get access to the registry, stores those inside of a file. And to be specific, the registry uses the Apache htpasswd format. This is a file format that's been around for a long time now and if you've ever worked with Apache, chances are you're already familiar with this. So that htpasswd file will contain a set of usernames and passwords to get access to the registry. And we'll create one in a minute, and then we'll need those username and passwords on the clients to access the images. So let's take a look at this in action.

Explaining the Basic Auth Example Files

Okay to work through this example of basic authentication, if you go into the course repo and you look in the security folder and then inside of the basic folder, there are a few files here that we're going to work with. One of these is a compose file, and then the other one is a Dockerfile. The compose file glues together the single service that we have here, a basic service, which is running a registry image. Now that's not apparent right away because this is building a custom image using the Dockerfile that's inside of that auth folder. We'll get to that in a minute. Now this is a registry, so we'll need to publish the registry API port, so in this case I'm mapping port 5000 onto 5001 on the host. And then I'm setting a few environment variables here to configure registry authentication. I'm specifying that I'd like to use htpasswd style authentication and then I'm setting a few parameters, one of which is the realm. The documentation is somewhat sparse on what exactly realm is. It maps to the WWW-Authenticate header, which is sent from the registry to unauthorized clients. It tells those clients how to authenticate, and we'll see this in a moment here. For the purposes of this course though, you can set realm to any value, you would only really set this to a specific value in some advanced authentication scenarios where maybe you want to authenticate between your registry and some external systems, it's not something we're going to talk about in this course though. I did want you to understand what this value is used for. And then after the realm, we specify the htpasswd file location. Now how did I come up with this path? Well, that's where the Dockerfile comes into play. So if I close the compose file and open up the Dockerfile, not a lot going on in here. We start from our registry image and I'm just layering in one additional command to make one slight change to the file system to create that htpasswd file. So I'm making the auth directory and then calling the htpasswd command, which is shipped with the registry image. I'm specifying that I'd like a user called admin and a password called admin. You can change those values if you'd like. You can even add additional users by calling htpasswd multiple times and appending the output to this file. And then I have a few flags specified here. To understand these, I'm going to pull up some help for the htpasswd command. So we've got little n just specifies to write to standard out, that way we redirect to the file. Little b says that we're going to provide the password at the command line. So we're not going to prompt the user to type in the password because we want this to be automate. And then finally, we've got big B here, this is specifying the type of encryption or hashing at play, which gives us some semblance of protection for the file that contains both the username and the password. It is possible, for example, to use plain text if you would like with the -p. Not a good idea in any production environment. Now that we've got that understanding in place, let's fire up this registry.

Basic Auth to the Registry API from a Browser

Okay so back over at the command line, what do I type in here to start all of this up? This is docker-compose, so I can call docker-compose up. I'll clear out the screen first here and start this up, and we can see the output then from building our custom image, and then also from starting up our registry. What could I do now to test out this basic authentication? Well how about we take a look at the catalog of images inside of our browser here. So I'll paste in localhost:5001, be careful I changed the port this time, and when I perform this request now in the browser, I'm prompted to provide a username and password, so our basic authentication seems to be working. Now I'm going to hit Escape here or the cancel button, that way I won't provide the username and password, and you can see when I do that then, I get an error message. So if you don't provide a username and password, you're going to get an unauthorized message now and you'll get that back as a JSON response, instead of our catalog of images. Also, if you take a look at the dev tools and go to the Network tab, refresh the page, hit Escape again to cancel login, and go to the failed request, and if you open up the response headers for that failed request, you'll see that www-authenticate header, that's a challenge that's telling us we need to use basic authentication. And there you go, you can see the realm that we set in our config file, that flows through to the header as well. And of course, that's how the browser knows what sort of authentication to use. We also have a 401 Unauthorized, so a proper HTTP status code. Alright, so if I reload the page again and this time if I provide the correct username and password, I probably won't remember that for right now, you can see we get back our catalog of images. So that's great. We've got our registry up and running and we can sign in now and without signing in, well, we don't have access. Okay so what else might we want to do with our registry now that we've got basic authentication set up? How about we push an image and see what happens.

Pushing Images to a Registry with Basic Auth

So we use the Docker CLI to push an image and I'd like to push out this busybox image to our newly protected registry. Can you tell me how to do that? First step is to tag our image so that we can associate it with our new registry that we would like to push it to. So we'll take busybox and tag is as localhost:5001. And then for the repository, let's use confidential. And next up, I can go ahead then and push the image. Now I've got a question for you, what do you think is going to happen when I push this image? Well let's find out. So we get an error back here. Actually sometimes it's not quite so obvious that we have a problem, but you can see the last line here says that there's no basic auth credentials available. So we can't push out to this registry because we don't have access to this registry. Obviously if we can't query it we can't push to it without also providing our credentials. What do you think I need to do here to provide the credentials? So if you've ever used Docker Hub to log in to push your own images, you've probably used the docker login command. This will also work with a private registry. If I grab some help here, you can see we can specify a server at the end of docker login. Now if you don't specify a server, you'll be logging into Docker Hub, but if we do specify localhost:5001 here, we're then prompted for a username and a password and now you can see we've logged in. After doing that, we can now try pushing our image again. Alright that looks successful, and if I come over to the browser and just refresh, there you go, we've got our confidential image pushed out to the confidential repository. Now before we move on, I want to show you a few things about how credentials are stored, something that peeked my curiosity and I think it'll better help you understand what all is involved when you're using credentials to access a registry.

Token Auth

We've now seen how we can lock down a registry so that only certain clients can access the images that are hosted. In this model we refer to as basic authentication because we are using that htpasswd file, and of course, in this model we have a username and password we're providing when we make a request to the registry. You have other choices though when it comes to security your registry. For example, you can use a token authorization scheme, and the best way to understand this model is thinking about splitting out that htpasswd file and putting it somewhere else. In fact, put it on a separate server or inside of a separate service at least. And make it someone else's problem to authenticate users and authorize access. So think of splitting that responsibility out, and that's a really nice thing because now your registry can focus on serving up images and also storing images, the things that it does well, and it doesn't have to worry about storing credentials and the security around that, it can delegate that responsibility out. It also doesn't have to worry about who has access to what. So when you split out an auth service, that means that a client that wants access to the registry is now going to start by going to the authentication service and providing whatever credentials it has. Now quick quiz, what type of credentials might it provide to this authentication service? It could actually be basic authentication as well. And that could be stored in an htpasswd file It could be stored in some sort of database somewhere. Who knows. The sky's the limit because this is a separate service so you have a lot of flexibility in this. It's also possible that OAuth is involved. So I mention these two options, basic auth and OAuth, and actually a third, anonymous authentication. These are all three choices you have right now available to you that will work with the docker login command that we saw in this module. And in the future, there may be new choices that will work with that docker login CLI, which behind the scenes is actually providing the credentials that you specify to the Docker engine or daemon itself so that it can pull images from your registry. Now if we step through this model, if a client wants to pull an image or push an image, the very first thing it's going to do, it's going to reach out to the authentication service, it's going to provide its credentials. Hey my name is Wes and my password is food. And it's also going to tell the service what exactly it would like to perform against a registry, like I might want to push a mongo image to the registry, so I'm going to specify that, as well as provide my credentials. So the credentials will authenticate me and if that passes, then the action that I want to perform will be used to authorize my access. And it's in specifying what you want to do that you can have an external service say yes or no. Yes, you're in the IT department so you can get to the IT registry or IT repositories within the registry, and no you're in HR so you can't have access to that. So all that's determined then by your authentication and authorization service that's external and it generates a token, sends that back down to the client. This token has information about the actions you requested and then it has a signature from the authorization service so that you can then, as a client, take that token, almost as if it was the basic auth user and password we worked with before, cram it into an authentication header and send it off to the registry and perform your action. So really if this is at all confusing, in this model, you're just sending a token instead of a username and password. Now how you get that, that's an entirely separate conversation and entirely separate course actually, if we want to get into standing up authorization services, the sky's the limit when it comes to picking one of those. But the gist is, you get this token and then you send that instead of a username and password, and the registry then just validates the token, it doesn't worry about keeping a database of credentials and who has access to what. Now two things, if you're interested in this model. First up, I would encourage you to look at the configuration section to set this up within your registry. You'll see there's not a lot to specify here. Very similar to what we have with basic auth. And down below, you'll find details about each parameter that you need to specify. The gist is, you need to provide a realm like before. The service is you as the registry or the repository of images or the resource when it comes to token-based authorization. Who's issuing the tokens and then also a certificate bundle for validating the signature on the token. You'll obviously know more about this in the context of your specific token server if you're going to use this model. There's also a great set of docs that explain how the token authentication works. This particular article is really good because it describes it in terms of a registry. It even has this nice visualization at the top of the workflow and a little brief description of that. And then beyond this page right here, over on the left-hand side in the table of contents, you'll find a total of five pages describing this authentication model. What I would like to do, so maybe you can wrap your mind around this a little bit more, a second thing I'd like to show you is what this looks like against Docker Hub. So let's take a look at using this workflow to hit the Docker Hub registry API. Let's do all that from the command line and take a look at these general steps to get a token and then pass it back to the registry.

Token Auth with Docker Hub

Alright I want to walk through a quick example of using token-based authentication and authorization with Docker Hub. And to help out, I've got a script called query. sh. This is in the security/hub folder in the course repo. I'm just going to execute this and step through what's coming out of this script to explain what's going on. After running that, I'll hop back up to the top here and we can walk through this output. So the first step that this script performs, it's just trying to query the set of tags for the alpine repository, the official alpine repository, out on Docker Hub. So this should look familiar to you, this v2/library/alpine and then tags/list. We used this earlier against our own private registry. I queried this first because I wanted to show you that you'll get back an unauthorized response and we're getting that because the Docker Hub registry requires token-based authorization. Scrolling down a bit, this is the web request that's made and this is the set of headers that come back from trying to query our tags. In the response here, in the headers, you can see we are being challenged to authenticate by providing a bearer token. And inside of here, you'll see details about where we need to go to ask for a token. Now the scope is the set of actions that we want to perform, in this case to pull information about the alpine repo. And the last piece here is the service or resource that we want to use. So we need to be authorized to use the Docker registry. So the next step then, number two here, is to get that token. Right below is the token that I received on this request. It's basically making a web request to the authentication service that's specified here, providing the details that it told me to provide, and then getting a token back with that information inside of it. So now what I want to do is copy the token and hop out to this jwt. io website and you can paste this in here and you can debug or break apart what's inside of this token. Now don't do this with a token that has access to a sensitive system, but for accessing Docker Hub, this is not an issue. If you scroll through here, you'll see the payload of this token contains an access section and you can see, it's got the same scope that we specified earlier. This is the set of actions that we would like to perform to pull from the alpine repository. Also has some information about where this token is valid at and the duration of the validity of this token, some dates, when it expires, when it was issued, who issued the token, etc. Down below, you can see if the signature is verified and just for fun you can change the signature section, which is the blue text, and see if the signature is still valid. The neat thing is, all of the information you send up that's a part of your authentication challenge, all of that gets mashed into this token, this token is signed, so we then take that token and hit the exact same tags API again, so step 3 here, we pass the token when we make that initial request, and bam we get back the list of tags that we are interested in. So we were working at a pretty low level here orchestrating each of the steps to interact with the API. I did that because if we hop up a level, the docker login command will take care of this for you to capture the credentials that you specify to be able to get the token and then it will take and deal with the token and passing that off to the registry for you so you won't see all of it, but if we look at making requests ourselves to the registry API, we can see what's going on. So I hope this illustrates how this works, just as you get a token from an external service and you use that as your password basically. And along with this model, you get all the goodness of separating responsibilities. So if you're interested in this model, a really good exercise now would be to stand up a testing auth service, similar to this auth. docker. io, and try and get it to work with your registry. So go to that configuration reference and get your configuration set up to plug the two together.

Silly Auth

There's one last model I want to point out for authentication and authorization and that is what's called silly auth. This is a model knit for testing only, not any sort of production environment, not even if you have a temporary problem with authentication and you just want to shut everything off. Because in this model, all the client has to do is provide an authorization header. The value of the header is irrelevant. If you're curious, in the security/silly folder, there is another example that you can docker-compose up. Once you have that up and running, try making a request to the registry API and include, for example, with curl, the -i flag to print out the protocol headers. When you make this request without the authorization header, you'll see that you are unauthorized. And if you make another request here and add the authorization header, and set the value to anything, when you make that request, you'll see you are authorized now and you get back your catalog of repositories. This can be useful in a testing environment when you want to ensure the header is there, but you want to disable validating what's passed along with the header.

You Can Use a Proxy Too

Well we have a couple of models here to choose from that are integrated right with the registry, when comes to authentication and authorization, to lock down client access. You should also keep in mind that you can stick with the time tested approach of sticking a proxy in front of your registry and letting it do all the heavy lifting of the extra functionality that you might like your registry to have, like securing access. Then your registry can really just focus on storing and retrieving images, and that's about it. And if you're interested in this, I would refer you to a few good pieces of documentation. So out in the docs under the Recipes section, there are two recipes to look at, one for Apache and one for Nginx.

TLS

Thus far in the module we've done a good job covering how we can secure the client-side of things, that way only the clients that we want to have access can push and pull images. Now I've got a question for you, when it comes to security, what else should we be concerned about? Well how about we turn things around and talk about securing the server side of things. How does a given client know that it's talking to the server that it thinks it's talking to? And then next, how about securing the medium of communication? How good is a password, or even a token, if somebody could sniff the network traffic and just snag your token or password and use it then to authenticate to your registry? We have to be concerned with the communication channel and also with the server itself, otherwise when we go to push a sensitive image, somebody could steal that image if they're impersonating our server. Now do you have any idea how we might address these problems? Well it turns out, if we simply generate a certificate for our server, for our registry, that the client trusts, we can then use that to encrypt the communications, as well as to verify that the registry is who it claims to be. So next, we're going to turn our attention to TLS. So instead of plain old HTTP, we're going to make sure that we're encrypting our traffic, and we're going to do it using a certificate that our client trusts.

TLS Example Files

So just to get our bearings, I've got the configuration documentation opened up here and I'm looking at the HTTP section, so configuring how we listen for traffic over HTTP. And as you can see, there are quite a few options to configure, one of which is TLS. So if you scroll down here, you'll see the tls section, and inside of here, the gist is we're configuring the certificate for the server, as I mentioned, we need to generate a certificate that the client trusts that we can use to encrypt our traffic and also to verify the server. Standard stuff here, so let's take a look at an example of this. If you open up the tls folder out in the security folder in the repo, you'll see a compose file, which we're looking at right here, and then also a Dockerfile. The Dockerfile has a multistage build to generate a self-signed certificate for the server, this will make it easier for you to follow along with this example and try things out yourself. So I'm using the openssl command line and I'm requesting to generate a self-signed certificate, and you can look at the parameters here and tweak them if you would like. Down below, I am then using a registry image and I'm just copying that certificate into the image. So this is just baking a certificate into the registry image, that's all it's doing. So we'll close that away and we'll focus on the compose file instead. So inside of here, we have one service defined, it's based on that self-signed Dockerfile, building an image from that. Open up a port 443 for SSL, instead of 5000, and because of that then we need to configure a few things. So I'm using environment variables again, you could use a config file if you'd like. I'm specifying under that HTTP section we just looked at, two separate configuration items, the address, so I'm binding the registry to listen to port 443 on all interfaces, and then also I'm specifying the TLS\_CERTIFICATE and the TLS\_KEY. And if you look here, you'll see these are pointed at the self-signed certificate that I just generated inside of the Dockerfile. So nothing fancy here, we're just creating that certificate I was talking about and plugging it into the configuration for the registry so it uses it then to serve up the registry API over TLS instead of just plain old HTTP. Join me in the next video and let's get this started up.

Encrypted Communications to the Registry API

Okay, so over at the command line, let's bring up this environment. So it'll start up our registry and first it will build the image, and you can see it's generating a certificate up there. Once it's done that, it'll start up our registry using that new self-signed certificate. And you can see in the output down below here, here we go, we are listening on port 443, and we are using TLS. So it looks like we are good to go. Now I've got a question for you, what could I do to validate that this is configured properly and we actually have secured communications? Well we have a couple of different vectors to approach this. How about we start with our browser and just make a request to the registry API. Alright, so I'm over in the browser, what do I type in here? Well I ask this because you should be careful here, we are flipping over to https, so make sure you specify that, and we're not using port 5000 anymore. We've published port 443, which is the default for https, so we don't need to specify a port, so just localhost works fine here. And then we have v2 and then \_catalog. Okay, and make my request there and you can see the connection is not private, we're getting this warning, and that's because our browser does not trust that self-signed certificate, and that makes sense, we just generated it, so our browser cannot validate if the person that claims to be localhost on port 443 is actually who we think it is. So that's that separate dimension of actually trusting our registry. So we have one thing in place, and that's encrypting our traffic. The second thing is making sure we verify that the server is who it says it is. For now, if I want, I can come under Advanced here and go ahead and proceed anyways, and you can see I get back my list of repositories here. Now if I had to push a button here to tell Chrome to go ahead and trust the certificate and finish making a request to the API and see my list of repositories here. I'm sure you can imagine then that we're going to have a little bit of trouble doing something else. Can you take a guess what that is? We're going to have trouble pushing and pulling images.

The Docker Daemon Needs to Trust the Registry Certificate

So normally by adding a certificate to our registry, we would be complete with both the task of encrypting communications and also getting the client to trust our registry, and the reason we'd be down with both is because we would have used a certificate that our client automatically trusts somehow. Likely using some trusted certificate authority, whether that's a public CA or perhaps even an enterprise CA that we maintain. We would not be using self-signed certificates in a production environment. So I could just skip this topic, but I thought it would be a good opportunity to show you how you can go about getting custom certificates to be trusted by Docker. And obviously a self-signed cert is a great example of a use case for that, but there are other cases where you might have a certificate that you would like to trust that for whatever reason is not automatically trusted by your Docker daemon. Now the first thing I'd like to do is point you at some documentation because the approach you take to trusting a certificate is going to be dependent on your environment, so on the operating system you're using. And it just so happens that I think this is the best documentation I can find of the official documentation, and it's under this Use self-signed certs section, which is what we're doing, so that's great. It's step 3 here. And under here you can see various different operating systems or different installation mechanisms for Docker, like Docker for Mac, Docker for Windows, with links to other documentation. So I would encourage you to refer to this, it's kind of like a top level overview of where to go. That said, this does not present quite all of the options because, for example, this certs. d approach with Linux is also applicable for the Docker for Mac and Docker for Windows approach. You'll find a. docker folder in your home directory, and inside of there is a certs. d folder, and so this approach can actually work too. So it's not all encompassing, I haven't quite found that in the documentation, but this is a good place to start and it'll give you some options. And I definitely would not recommend memorization of these techniques, instead, bookmark these docs for future reference. Alright, so the first thing I want to do is hop over to the CLI here and just show you the failure that we'll get if we try to push an image to our TLS secured registry. So first thing here, let's go ahead and tag our name and image so that we can push it to our registry. So I'll tag this with localhost/busybox and then I can go ahead and push that image. And strangely enough, the image pushes to our registry. We can verify this in the browser. So you can see, we've got our busybox image pushed out. Now that's strange, I just said that we would have issues. What's going on here? Why is our Docker daemon trusting the certificate? Well it turns out the daemon is not really trusting the certificate. If I come back over to the CLI, if I run the docker info command, and I'm only going to show the insecure registries here in the docker info output. When I do that, you can see the registries that are bound to the loopback interface, 127. 0. 0. 1, are going to be treated as an insecure registry, that means that the Docker daemon is going to ignore any certificates. So we could use an IP address on the interface on my Mac that is external to the machine so that we're not using 127. 0. 0. 1 anymore. Instead, I've set up example. com in my host file, I set that up to point at my Mac's external Ethernet card. So if we were to try and push an image to this registry now, which is the same registry, but via a different interface, we should have trouble then. By the way, if you're on Windows or Linux, you can set up this entry in your host file as well, if you'd like to follow along. So let's go ahead and take that busybox image again, this time tag it with example. com/busybox, which is the exact same registry, just a different way to get to it. And then I can go ahead and push an image at example. com image, and now we have an issue. We have a certificate signed by unknown authority. So now we've got a problem and we've got something that we can fix. Let's take a look at this next.

Pushing Images Fails with Untrusted Certificate

Alright, so this error makes sense now. We've got a self-signed certificate and our Docker daemon does not trust it to pull and push images. So it's preventing our image push in this case. By the way, if you really want to prove that, you can stop your registry and you can run docker-compose up, again with the -V flag, this will reset the anonymous volumes and make sure there are no images and repositories out in your registry, so it will clear out the registry. And when you refresh in the browser, now that original busybox image, that we pushed via localhost, won't be there and you can verify that this image push is actually failing. You can see we still don't have that repository in our catalog. So now we can set up trust with the certification for our registry.

Trusting a Self-signed Certificate

So to fix this problem, we need to get access to the certificate. And an easy way to do that is to use the docker cp command to copy files out of the file system of our running registry container. We can copy out that self-signed certificate that we generated. And if you have completion set up here with the command line, you can type in tls, which is a part of our compose project name, hit Tab and you should complete then to the one container that you'll want to use to copy out the certificates, and it's at /certs is where we copied that in, and I can just specify that I'd like to copy that into the current directory. So this will bring this back to my host machine where I can access it, and you can see we now have a certs folder. If I look inside of the certs folder, you can see our two self-signed files, the certificate and the private key. We need the certificate. We don't actually need the private key, so you don't have to copy that out. If you are a graphically inclined person, double click the cert file, select the system keychain, and go that route to add or trust the certificate. I like the command line, so I'm going to go that route instead. And of course, this is specific to a Mac, I'm using the security command, calling a subcommand called add-trusted-cert, this will do the same thing as double clicking that file. Actually it'll do a couple of things simultaneously, it'll add the certificate and it'll set up trust. I'm setting this up with the admin user in the system keychain. You can also add this to your login or per user keychain, so it's not available globally. And then I'm specifying that selfsigned. crt file that we just copied out. Put in my password and that will have added our certificate to my system keychain and Docker for Mac turns to the keychains you have to determine what certificates to trust. So now let's test out pushing again, and I've got a question for you, what do you think is going to happen? So I just want to try pushing again to example. com, and you'll see we still have an issue, the certificate is still not trusted. Can you take a guess why that is? Well we added that certificate to the OS X keychain, the system keychain, but we have not done anything to tell our Docker daemon that there's a new certificate that we've configured. So it's important that you restart the daemon, in my case, restart Docker for Mac. So I can come up here and click Restart in the menu, give that a second, you'll see the icon flashing, and when it completes we should be good to go. Now before I push again, I just want to show you that the registry is indeed empty. You can see nothing in here. If we want, we can even load up, instead of localhost, we could point at example. com, and there you go, we've still got an empty set of repositories. So if I come over to the command line now and hit docker image push, what do you think will happen? Well here we go. Take a look at that, it works. So our registry is now trusted with the certificate that we added. Both the certificate that we had to add to the registry, but then also we had to tell the Docker daemon to trust that certificate. Now normally you might not have to do that, if for whatever reason your chain of trusted authorities already has your CA that you're generating certificates for, for your registries, which is probably the case for a lot of you, but if you happen to have a certificate that needs to be trusted you can follow this process to go about trusting it. One last time, I want to reiterate that an alternative to using the keychain on a Mac is to use this certs. d folder. Now on a Mac, if you come in under adding custom CA certificates, that certs. d folder is in your home directory in. docker and then certs. d, and then inside of the registry directory under that, so you'll have one directory for each registry, you can drop in that same server certificate that we just added to the keychain and this will also work. So it's an alternative to the keychain. And separately, I am highlighting an explanation of how to add client certificates, which are optional and can be added on top of the server certificate. And with this then, we've secured the three pieces that we are interested in securing.

Removing Trust in a Self-signed Certificate

Before we wrap up, let's take a look at getting rid of that certificate that we trusted because it's probably not a good idea to leave this in the list of your trusted certificates. So if you come into keychain, in this case, you can remove the trust by just deleting the certificate. It'll prompt you a few times for your password. On Mac OS there are two parts, there is a certificate and then the trust settings that are set up for what you're exactly allowed to do with that certificate. So both of those are now removed, and if you come back to the command line, question, what will happen if I try to push an image now? Well here we go. It works. Why is that? That's because we have not restarted the Docker daemon, so we need to do that. And then when we push an image, now we get our error again. So that's how you can clean up after testing a self-signed certificate. We've now reached the end of the module, along the way we saw how we can secure image distribution from the client through the networks, and finally to your registry server. Of course this is just a foundation of security. I would strongly encourage you to look at related topics, for example, verifying who built an image. You might look at a tool called Notary to help you with that. Because what good does it do us to securely distribute images if we don't know who built them in the first place? Maybe they came from a bad actor. Another tangential topic is vulnerability scanning, because even if you know who built an image, over time vulnerabilities can be exposed. And if you're standing up your own registry to house your own images, then you might have one spot you need to go now and one consistent format, that of an image, to be able to easily scan all of your software for vulnerabilities. That's a really exciting thing, that's a great value proposition for packaging your software up as images. And I'm showing Clair here because it's a piece of software that might help you accomplish this. For now, let's turn our attention to some of the topics around deploying a registry into a production cluster environment.

Preparing a Production Cluster Registry

Shifting Attention to Production Servers

Throughout this course we've covered many aspects of running your own registry. We've even considered the perspective of various different clients that might access our registry to pull images. Now I'd like to turn our attention to a particular subset of clients. I'd like us to focus on servers. It's not that their concerns are unique, but I think a common use case for a private registry is to be able to house your own internal images to distribute those in a production environment to run your internal applications. Those tend to be proprietary, so hey, a good use case for standing up a private registry. Of course, we've already covered some of the topics that matter when it comes to standing up a sensitive internal registry, especially one used in a production environment. We've touched on security, for example, we've touched on different storage back ends, we've even talked about notifications that can integrate multiple registries together that are controlled by separate teams. So bring all of that perspective with you to this module, and now let's turn our attention to a few more considerations when it comes to running a registry in your production environment where you probably have quite a few machines that are running probably quite a few containers that need to pull images from your own private registry.

Network and Node Level Registry Access

First, I'd like to talk about security, specifically within the context of a production cluster. So let's take the ideas from the previous module and let's apply them just to this particular situation and explore some of the nuances that we did not touch on. And I would like to start by saying, if you have a production environment that's highly isolated, perhaps a private network, maybe even an air gapped network, there's a good argument to be made that you don't need to lock down your registry at all if you've already locked down the set of nodes within that network so that they're the only nodes that can access your registry. In a way you could think of this as a network level of access as your security boundary. It's a perfectly fine deployment model if you can guarantee that your network is secured. And I want to explicitly point that out because all too often I think people cringe when they hear the idea of leaving some sort of service unprotected, but the reality is, if you can't secure your network, if that's your intent, then what makes you so certain that you can secure an individual machine or safely distribute some sort of credentials to access a registry? If you can't don one with any level of certainty, then you probably can't do any with any level of certainty. Nor can you do any of these with 100% guarantee that nothing will go wrong. So the network level of access is a first option. If, however, that doesn't work for your situation and you want to lock down your registry, you could use any of the techniques we talked about in the previous module. It doesn't really matter for our context here. The one thing we do need to talk about is how we distribute the credentials to be able to access that registry. That could be a username and password, it could be a token, could be any sort of authentication that you have set up, maybe with some external service to the registry. It doesn't really matter. The specifics of how you deploy these credentials is up to you. Chances are, you'll integrate it with your configuration management tooling or processes. So whatever you're doing right now to bootstrap a node to install the necessary software on it, you might consider adding something to be able to access the credentials then to get to the registry, that way the credentials, you could just think of them as sitting there on the machine so that any application that runs or any container that runs on a given node can have access then to images on that registry, or I should say, can be run based on images on that registry. So that's one approach, just give the entire node access to pull images from the registry. Or a slight variation would be to only give a subset of your nodes access and then when you're deploying your running containers, only deploy those to the nodes that would have the credentials that they need to be able to pull the image for this application that's stored on your sensitive registry. Again, configuration management of some sort that stands up a given node is going to be the route to approach injecting those credentials into that node's environment. And of course, keep in mind this is a cluster of production compute capacity. It's very possible you have other applications running that are not based on images stored on your own private registry. In that case then, you don't have to worry about the credentials because maybe those are being pulled from Docker Hub. For example, maybe you have NGINX running inside of your production environment. Maybe that's based on the official NGINX image with no modifications.

Indirect Access via the Node Level Docker Image Cache

There is a slight modification we can make to this design whereby we are essentially granting each node access to the registry, so that anybody that can get on that node and run Docker commands can start up containers. Of course, it probably wouldn't be people, it'd be probably be some sort of orchestration software, but at the end of the day there's another way we can approach this design without storing the credentials on a given node forever. Now before I spell this out, I'd like you to take a stab at it. What could we do so that we can use our private registry images on a given node, but not need to keep the credentials on that node permanently? Certainly there is more than one right answer to this question. If you came up with something unique, maybe share it in the course discussion. But for right now what I'd like to talk about is a design where we don't store the credentials permanently, but instead we take advantage of the Docker daemon's image cache on each node. Sometimes this isn't obvious as a design possibility when it comes to scaling out a cluster of machines, but hey, Docker images are huge and so we naturally have this means of caching them built into the daemon. Just imagine if every time you wanted to run a container based on an image, imagine if you had to pull that image each time. Obviously it wouldn't be a very pleasant experience, Docker would not have taken off, and we wouldn't be where we're at today, and you probably wouldn't be watching this course. So thanks to this image caching, all it takes is pulling an image one time, even if it comes from a registry that's secured, once it's available on the image cache, anybody else on that machine can start up a container based on it. So if for any number of reasons you don't want to put the credentials on a node, for example maybe you're testing out a new registry that you just set up, and you don't quite have your credential story figured out for distributing access across your cluster, well hey, hop on the node, log in, pull the image, log out, and then it's going to be there for other people. If you like the idea of using the image cache instead of storing credentials on all the nodes, then you're going to be concerned with how you go about synchronizing those images with the images on your registry. And there's no right answer, there's no one way to do this for sure. And in the spirit of creativity, can you take a minute and think of one way that you can keep the image cache on a given node in sync with your private registry? So I hope you didn't come up with a solution of having the intern go into each machine, SSH in and run Docker pull on a few images. Certainly that would work, but that wouldn't be very fun for them. However, you could base a solution to this problem on that idea of a job running periodically. You could have, for example, a cron job that runs maybe every 15 minutes and pulls down a set of images off of your registry. You could even query the catalog to determine what images to pull. Now you'd still be stuck with the challenge of figuring out how to get the credentials to that cron job, but then you could limit the credentials to that simple cron job that runs instead of needing to make them available to any time somebody launches a container. So that's one possible solution. Another one, based on registering notifications, and maybe this came to mind since we covered those in this course, when the registry receives a new image, a notification then could kick off some sort of process that essentially makes a push base model out of pushing images then into the caches on all the various different nodes.

Linking Registry Credentials to a Service Definition

Another approach for giving the credentials to pull images is to tie that set of credentials to some sort of service definition that you pass off to an orchestration tool, for example, maybe Docker swarm mode or Kubernetes, to pass off a service definition that links to credentials that can access the image that you need for that service to operate. So essentially when you start up an application, you're going to be tying the credentials to that application. Then you could run that app anywhere because the credentials will be provided with your application definition to be able to pull the images needed just maybe for your application and nothing else. This may actually be the easiest approach going forward, at least of the approaches I've covered, simply because this doesn't necessarily require any special configuration on a given node. Your orchestration software that controls all of your nodes, that's where you would set this all up, both the application that you're going to run in your service definition, as well as the credentials that are needed. And to be specific, the credentials, because they are sensitive, are often stored in what's known as a secret. Let's quick take a look at an example of a secret. Here when I'm working with Kubernetes, I've displayed some help here for a command that will create a secret, and you can see one of the available subcommands is to create a secret for a docker-registry. So this notion is built right in, it's baked right in as a first class concept. Obviously because if you're going to be spinning up containers with Kubernetes, probably going to need to occasionally have access to a private registry. And of course, if you pull up some more help for this command, and if I scroll through here, you'll see a section where you can provide username and password, for example, as well as the registry server. These credentials then get linked to ultimately containers that are going to operate on your cluster, and then as the container orchestrators spin up these containers, they're going to pull images based on these credentials. It's all distributed for you so you don't have to think about it, it's a really nice thing.

Scaling Registry Capacity

Moving on from talking about access to the registry, when it comes to a production environment, chances are you might get to a point where you have so many servers hitting your registry that you need some ability to scale your capacity. Probably going to be on the read side of things, unless you're building a lot of images frequently. Now if you want to scale up like this, there's one really important consideration that will limit your ability to do this. Can you take a guess what that is? Well if you want a set of registry instances to serve up the same set of images and also persist images to the same location, then you'll need to consider your storage back end. By default, if you spin up three separate registries, they'll each have their own file system storage isolated, so they won't be able to share images, which means it's probably not exactly what you're looking for in a scenario where you scale up your capacity. So the default storage driver that uses the filesystem that maps back to the host, most likely, unless you've done something special with volumes, that's not going to cut it for scaling. If I ever use something like the S3 storage driver, you'll be able to spin up additional instances and serve up the same set of images. So while it's important to consider your storage driver, I'd also say that I wouldn't jump the gun and worry too much about this from the get go, I'd worry about it when it's actually a problem because you might not have a need for scalability, thanks to the Docker image caching on the given nodes in your cluster. So don't solve this problem too soon, it's just something you'll need to consider, and when you're considering it, if the need arises, then you might want to look at migrating your storage driver to store your images somewhere else so that you can scale up. In addition to the driver, under the configuration for http on the registry, there is a configuration option for a secret. Normally this is generated for you, and you can see some more details here in the table about this. Normally this is generated for you, but if you're going to spin up multiple instances and load balance requests across them, you'll need to make sure that this value is the same on your registry instances. And of course, this touches on a separate concern that I don't want to get into, but load balancing. You need to have some means of load balancing requests, the good thing is, orchestrators these days, like Kubernetes and Docker swarm mode, they come with baked in capabilities to take care of load balancing for you.

Keeping Up

There are two things I want to accomplish in this video. As we approach the end of the course, I like to provide some additional resources and point out ways that you can keep up-to-date with changes in, well, whatever topic I'm covering in a course. So in this video and the next, I'll be covering those two aspects, and specifically in this video, I want to talk about additional resources, and the first resource I want to reiterate, because we've looked at this already in the course, is that the registry is open source. I would really encourage you to check out this docker distribution repository on GitHub, and specifically take a look at the Issues. If you're curious about what's coming in the future, if you'd like to request a feature, or just see what features are being discussed and maybe contribute to that discussion, you can come out to the Issues list here. You can even submit any problems you're having or look for people that have solved problems that you're having. Now there's quite a bit in this list of issues. I wouldn't recommend necessarily reading through all of these, instead search as you perhaps are looking for something specific and then if you would like to read through something, I think what is worthwhile is to look at the Milestones that are assigned to these issues. So we've got both 2. 7 and 2. 8. So here are some of the issues for 2. 7. You can see a discussion here of support for oci manifests and manifestlists, or work being done on letsencrypt support for TLS. This issues has to do with a bug right now, if you delete an image and then try to push it again without restarting your registry, there's a cache invalidation bug right now. This is something you might encounter, so this might be worth looking at. Another one that sticks out to me is this new tags API, keep an eye on this. And down below, it looks like perhaps some work being done to improve the documentation when it comes to deleting images, which also may be helpful. So that's 2. 7. There's also 2. 8, and one issue in 2. 8 in particular is private registry caching. This ties back to running a production cluster. You may want to set up your own registry caches or mirrors of your own private registries, and if you're interested in that, well, this issue has a lengthy discussion about how to add that support. This discussion has been taking place for quite a long time now and toward the bottom, you'll even see a link to a pull request with additional discussion, including some comments from about 11 days ago, as of the time of recording. Furthermore, when it comes to keeping up, at the root of this repository, there's a changelog. So you can scroll through here and see a history of the features that were developed in the current version of the registry. Sometimes that historical context can help you understand what might be coming in the future. Additionally, at the root of this repository is a roadmap markdown document. This includes some of the high level goals of the distribution project, as well as some high level discussions of the features that a registry should support, including some of the features that are still being worked on that may come in a future version of the registry.

Additional Resources

The last thing I'd like to do is point you at some additional resources that you can refer to, to go beyond what we covered in this course. And the first of these is the documentation, and I referred to this a few times to try to give you some pointers into the documentation for some of the topics that we discussed. I really think it's worth your time to read through all of the registry documentation. First we have some overview pages that are more high level features, these first five here, then we have some recipes, and you could look at these as you need to, if you want to try some of these things out. For example, we have proxying with Apache or Nginx, mirroring Docker Hub, garbage collection. So you could just come into these features as you need them, or as you use them, and read through these for a refresher or to learn a little bit more than what I covered with you in this course. And then down below this, a few more resources that are also worth reading. One of which, the registry API reference, the specification, this is definitely worth your time to come look at, specifically the Introduction and Overview. This is a very long set of documentation at 126 minutes, so don't read it all, but the first two parts, through the Overview, don't take that long to read through. And then the Detail and beyond, you can refer to this if you want to know the nuances of the registry API. Another area that's helpful for further learning stems from the fact that the registry is open source and there's a huge ecosystem around Docker and even the registry itself. And through this course, we used a couple of front ends that were not an official part of the registry project. We only used a couple of what's available, and if you don't like the features available in these front ends, I would really encourage you to look around for other front ends that you might want to use, or other stacks that you might want to use to integrate with a private registry, for example, Portus. There's a rich ecosystem out there. There may be a front end that better suites your needs. And so take some time to search around on the internet and find some of the articles where people discuss setting up some of these third party front ends and using them with a private registry. And to help you find some of these resources, check out the list of Docker resources available in this awesome-docker repo. This is a curated list of resources and projects in the Docker ecosystem, for example, you can find Portus, that I just mentioned. And if you search for a registry, you'll find quite a few hits on additional resources related to running a registry. Here are some front ends, for example. And there's actually a whole section dedicated to registries. And of course, don't forget to come pull the sample code for this course, try some of these example scenarios that are set up here. Many of them are just a simple docker-compose up, and you'll have a whole environment set up to experiment with one of the features of the registry that we talked about in this course. And then if you're curious and would like to learn more about me, I have a blog, and inside of my blog I have a contact page where you can reach out if you have some particular question. And of course, use the discussion page for this course if you have questions that you think other people would benefit from seeing the answers to. Or use the discussion page if you find a really neat resource, like a really neat front end that you appreciate, put it out there for other people to try, or if you have a problem with a registry and you solve it, leave a little note in the discussion tab so other people can see that. And speaking of which, if you make some code changes or maybe write your own scenario on top of the ones that I've shared in my GitHub repository, feel free to open up an issue on the repository and describe the scenario that you worked on, you can even open up a pull request if you think it would be helpful for other people to have that scenario available. And with that said, we've reached the end of the course. I hope you enjoyed it and I hope you're learning from it. The best way to lean is to get your hands dirty, look under the hood, deploy your own registry, try all off this stuff out. And I hope to help you out again in one of my future courses.