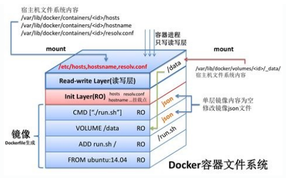
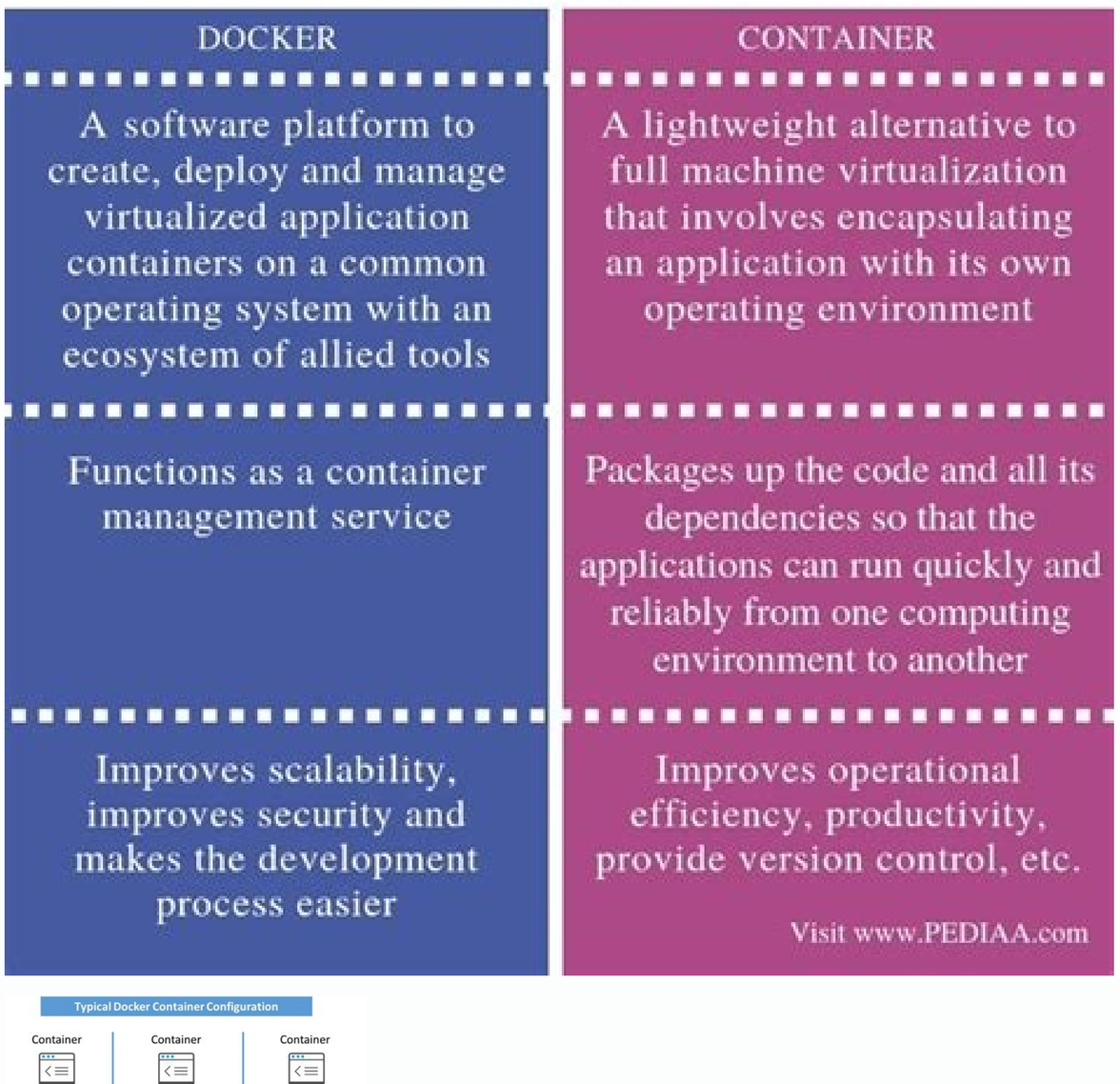
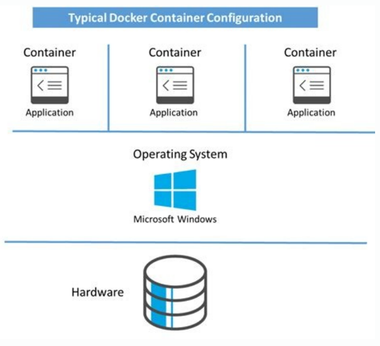
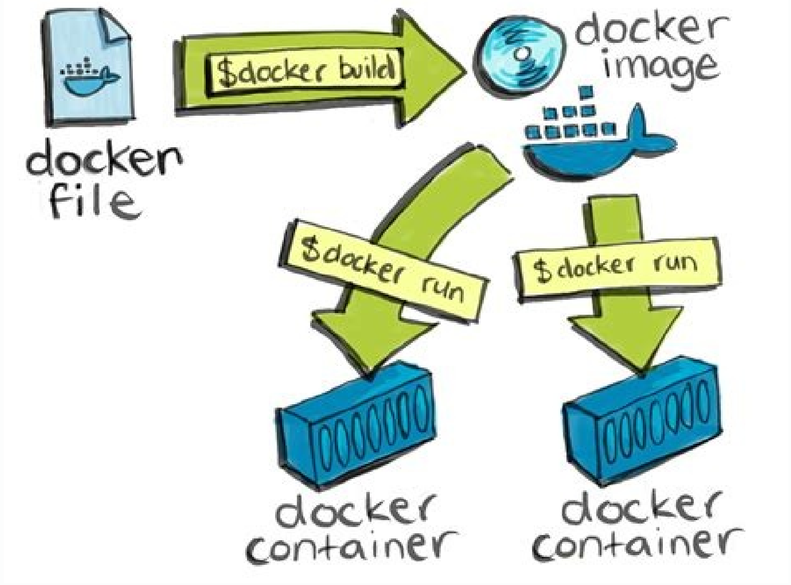
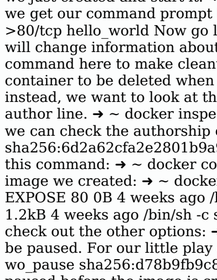
Docker file from container











Create docker file from container. Docker file from container to host. Docker remove file from container. Docker file from container. Copy file from docker container. How to get docker file from container. Copy file from docker container to local machine. Copy file from local to docker container.

SearchAppArchitecture SearchSoftwareQuality SearchCloudComputing SearchSecurity SearchAWS Estimated reading time: 5 minutes For the rest of this tutorial, we will be working with a simple todo list manager that is running in Node.js. If you're not familiar with Node.js, don't worry. No real JavaScript experience is needed. At this point, your development team is quite small and you're simply building an app to prove out your MVP (minimum viable product). You want to show how it works and what it's capable of doing without needing to think about how it will work for a large team, multiple developers, etc. Get the app Before we can run the application, we need to get the application source code onto our machine. For real projects, you will typically clone the repo. But, for this tutorial, we have created a ZIP file containing the application. Download the App contents from the getting-started repository. You can either pull the entire project or download it as a zip and extract the app folder out to get started with. Once extracted, use your favorite code editor to open the project. If you're in need of an editor, you can use Visual Studio Code. You should see the package.json and two subdirectories (src and spec). Build the app's container image In order to build the application, we need to use a Dockerfile. A Dockerfile is simply a text-based script of instructions that is used to create a container image. If you've created Dockerfiles before, you might see a few flaws in the Dockerfile below. But, don't worry. We'll go over them. Create a file named Dockerfile in the same folder as the file package.json with the following contents. # syntax=docker/dockerfile:1 FROM node:12-alpine RUN apk add --no-cache python2 g++ make WORKDIR /app COPY . . RUN yarn install --production CMD ["node", "src/index.js"] EXPOSE 3000 Please check that the file Dockerfile has no file extension like .txt. Some editors may append this file extension automatically and this would result in an error in the next step. If you haven't already done so, open a terminal and go to the app directory with the Dockerfile. Now build the container image using the docker build command. $ docker build -t getting-started . This command used the Dockerfile to build a new container image. You might have noticed that a lot of "layers" were downloaded. This is because we instructed the builder that we wanted to start from the node:12-alpine image. But, since we didn't have that on our machine, that image needed to be downloaded. After the image was downloaded, we copied in our application and used yarn to install our application's dependencies. The CMD directive specifies the default command to run when starting a container from this image. Finally, the -t flag tags our image. Think of this simply as a human-readable name for the final image. Since we named the image getting-started, we can refer to that image when we run a container. The . at the end of the docker build command tells Docker that it should look for the Dockerfile in the current directory. Start an app container Now that we have an image, let's run the application. To do so, we will use the docker run command (remember that from earlier?). Start your container using the docker run command and specify the name of the image we just created: $ docker run -dp 3000:3000 getting-started Remember the -d and -p flags? We're running the new container in "detached" mode (in the background) and creating a mapping between the host's port 3000 to the container's port 3000. Without the port mapping, we wouldn't be able to access the application. After a few seconds, open your web browser to . You should see our app. Go ahead and add an item or two and see that it works as you expect. You can mark items as complete and remove items. Your frontend is successfully storing items in the backend. Pretty quick and easy, huh? At this point, you should have a running todo list manager with a few items, all built by you. Now, let's make a few changes and learn about managing our containers. If you take a quick look at the Docker Dashboard, you should see your two containers running now (this tutorial and your freshly launched app container). Recap In this short section, we learned the very basics about building a container image and created a Dockerfile to do so. Once we built an image, we started the container and saw the running app. Next, we're going to make a modification to our app and learn how to update our running application with a new image. Along the way, we'll learn a few other useful commands. get started, setup, orientation, quickstart, intro, concepts, containers, docker desktop In this article, I'll provide step-by-step instructions on how to create a Docker container, modify its internal state, and then save the container as an image. This is really handy when you're working out how an image should be constructed because you can just keep tweaking a running container until it works as you want it to. When you're done, just save it as an image. You can use the following guide to customize and deploy the DataSet agent for common tasks such as adding parsers for your log files, adding redaction/sampling rules, running custom plugins, or mounting volumes for application log files. Before we jump right into it, we'd like to invite you to a relevant webinar on how to get the most value out of your Kubernetes Audit logs. If you run Docker containers in K8s environments, we will cover the best practices to implement comprehensive, secure, and efficient audit logs in production Kubernetes clusters. We look forward to seeing in the webinar. Step 1: Create a Base Container Let's get started by creating a running container. So that we don't get bogged down in the details of any particular container, we can use nginx. The Docker create command will create a new container for us from the command line: ~ docker create --name nginx_base -p 80:80 nginx:alpine Here we have requested a new container named nginx_base with port 80 exposed to localhost. We are using nginx:alpine as a base image for the container. If you don't have the nginx:alpine image in your local docker image repository, it will download automatically. When this happens, you will see something like this: Unable to find image 'nginx:alpine' locally alpine: Pulling from library/nginx df9b9388f04a: Pull complete 5867cba5fcbd: Pull complete 4b639e65cb3b: Pull complete 061ed9e2b976: Pull complete bc19f3e8eeb1: Pull complete 4071be97c256: Pull complete Digest: sha256:5a0df7fb7c8c03e4158ae9974bfbd6a15da2bdfdeded4fb694367ec812325d31 Status: Downloaded newer image for nginx:alpine 85b13f4d8a9bcdab4fbae540cf7bf3704eab13b57c5f44a2d3529d86f1c72ba5 Step 2: Inspect Images If you look at the list of images on your system, you will now see the nginx:alpine image: ➜ ~ docker images -a REPOSITORY TAG IMAGE ID CREATED SIZE amitsharma/nginx-reverse-proxy v1 1037dc5f8db4 3 weeks ago 142MB nginx-reverse-proxy latest 1037dc5f8db4 3 weeks ago 142MB amitsharma/web-server-app v1 09a0abf08e08 3 weeks ago 58.3MB web-server-app latest 09a0abf08e08 3 weeks ago 58.3MB nginx alpine 51696c87e77e 4 weeks ago 23.4MB Step 3: Inspect Containers Note here that the container is not running, so you won't see it in the container list unless you use the -a flag (-a is for all). ➜ ~ docker ps -a CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES c365af6303e4 nginx:alpine "/docker-entrypoint.…" 6 minutes ago Created nginx_base Step 4: Start the Container Let's start the container and see what happens. ➜ ~ docker start nginx_base nginx_base Now visit with your browser. You will see the default "Welcome to nginx!" page. We are now running an nginx container. Step 5: Modify the Running Container So if you wanted to modify this running container so that it behaves in a specific way, there are a variety of ways to do that. In order to keep things as simple as possible, we are just going to copy a new index.html file onto the server. You could do practically anything you wanted here. Let's create a new index.html file and copy it onto the running container. Using an editor on your machine, create an index.html file in the same directory that you have been running Docker commands from. Then paste the following HTML into it: Hello World Hello World! Then save the file and return to the command line. We will use the docker cp command to copy this file onto the running container. ➜ ~ docker cp index.html nginx_base:/usr/share/nginx/html/index.html Now reload your browser or revisit . You will see the message "Hello World!" in place of the default nginx welcome page. Step 6: Create an Image From a Container So at this point, we've updated the contents of a running container and as long as we keep that container around, we don't need to do anything. However, we want to know how to save this container as an image so we can make other containers based on this one. The Docker commands to do this are quite simple. To save a Docker container, we just need to use the docker commit command like this: ➜ ~ docker commit nginx_base sha256:0c17f0798823c7febc5a67d5432b48f525320d671beb2e6f04303f3da2f10432 Now look at the docker images list: ➜ ~ docker images -a REPOSITORY TAG IMAGE ID CREATED SIZE 0c17f0798823 About a minute ago 23.4MB amitsharma/nginx-reverse-proxy v1 1037dc5f8db4 3 weeks ago 142MB nginx-reverse-proxy latest 1037dc5f8db4 3 weeks ago 142MB amitsharma/web-server-app v1 09a0abf08e08 3 weeks ago 58.3MB web-server-app latest 09a0abf08e08 3 weeks ago 58.3MB nginx alpine 51696c87e77e 4 weeks ago 23.4MB You can see there is a new image there. It does not have a repository or tag, but it exists. This is an image created from the running container. Let's tag it so it will be easier to find later. Step 7: Tag the Image Using docker tag, we can name the image we just created. We need the image ID for the command, so given that the image ID listed above is 0c17f0798823, our command will be: ➜ ~ docker tag 0c17f0798823 hello_world_nginx And if we look at the index of images again, we can see that the s were replaced: We can actually use complicated tags here with version numbers and all the other fixings of a tag command, but for our example, we'll just create an image with a meaningful name. Step 8: Create Images With Tags You can also tag the image as it is created by adding another argument to the end of the command like this: ➜ ~ docker commit nginx_base hello_world_nginx This command effectively commits and tags at the same time, which is helpful but not required. Step 9: Delete the Original Container Earlier we started a Docker container. We can see that it is still running using the docker ps command. ➜ ~ docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES c365af6303e4 nginx:alpine "/docker-entrypoint.…" 33 minutes ago Up 25 minutes 0.0.0.0:80->80/tcp nginx_base Let's stop and remove the Docker container that is currently running and delete it. ➜ ~ docker stop nginx_base nginx_base ➜ ~ docker rm nginx_base nginx_base If we list all of the Docker containers, we should have none: ➜ ~ docker ps -a CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES Now, let's create a new container based on the image we just created and start it. ➜ ~ docker run --name hello_world -d -p 80:80 hello_world_nginx 7ca08e03862dcdaba754718be2fef18b8f9c57291fe25da239bd615a7802a80f Note that docker run is the equivalent of executing docker create followed by docker start; we are just saving a step here. The -d option tells Docker to run the container detached so we get our command prompt back. Step 10: Look at Running Containers If you look at the running containers now, you will see we have one called hello_world: ➜ ~ docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 7ca08e03862d hello_world_nginx "/docker-entrypoint.…" 2 minutes ago Up 2 minutes 0.0.0.0:80->80/tcp hello_world Now go look at . As you can see, the index.html page now shows the "Hello World!" message just like we wanted. Stop the container hello_world before moving on to the next section. ➜ ~ docker stop hello_world hello_world Step 11: Consider Your Options There are a few optional things we can do using the commit command that will change information about our images. For example, we might want to record who the author of our image is or capture a commit message telling us about the state of the image. These are all controlled through optional parameters to the commit command. Let's go back to our original running container. We are going to use a slightly different command here to make cleanup easier: ➜ ~ docker run --name nginx_base --rm -d -p 80:80 nginx:alpine 92158632375ff41c0232cf69fcdff7beeec981ffd552eaaef11f45db3f061da3 This command will run the image nginx:alpine with the name nginx_base; the creation of the image will be included in the command execution. The –rm will cause the container to be deleted when it is shut down. The -d tells the command line client to run in detached mode. This will allow us to run other commands from the same terminal. So if you visit now, you should see the default nginx welcome page. We went through changing things about the running container above, so I won't repeat that work here; instead, we want to look at the various options around the commit sub-command. Option A: Set Authorship Let's start by setting the authorship of the image. If you inspect the docker image hello_world_nginx above, you will discover that its author field is blank. We will use the docker inspect command to get the details of the image and grep out the author line. ➜ ~ docker inspect hello_world_nginx | grep Author "Author": "", So if we use the author option on the docker commit command, we can set the value of the author field. ➜ ~ docker commit --author amit.sharma@sentinelone.com nginx_base authored sha256:d0229f7f014bc510c16ec03d3c9ebcf25594827fde18c274cea2f44d116c948e And we can check the authorship of that image: ➜ ~ docker inspect authored | grep Author "Author": "amit.sharma@sentinelone.com", Let's delete that image and try some other options: ➜ ~ docker rmi authored Untagged: authored:latest Deleted: sha256:d0229f7f014bc510c16ec03d3c9ebcf25594827fde18c274cea2f44d116c948e Deleted: sha256:6d2a62cfa2e2801b9a9e5ed0a5ccf5e173e621e7fe05b15f32379537464e38ec Option B: Create Commit Messages Let's say you want a commit message to remind yourself what the image is about or what the state of the container was at the time the image was made. There is a –message option you can use to include that information. Execute this command: ➜ ~ docker commit --message 'this is a basic nginx image' nginx_base mmm sha256:d717f5e1285ec7a539f1e59908375ef3111f59f176ec0e40ec5835ddc96d9816 Using the image name, we can look at the history of the Docker image to see our message. Here we are using the docker history command to show the change history of the image we created: ➜ ~ docker history mmm IMAGE CREATED CREATED BY SIZE COMMENT d717f5e1285e About a minute ago nginx -g daemon off; 1.09kB this is a basic nginx image 51696c87e77e 4 weeks ago /bin/sh -c #(nop) CMD ["nginx" "-g" "daemon… 0B 4 weeks ago /bin/sh -c #(nop) STOPSIGNAL SIGQUIT 0B 4 weeks ago /bin/sh -c #(nop) EXPOSE 80 0B 4 weeks ago /bin/sh -c #(nop) ENTRYPOINT ["/docker-entr… 0B 4 weeks ago /bin/sh -c #(nop) COPY file:09a214a3e07c919a… 4.61kB 4 weeks ago /bin/sh -c #(nop) COPY file:0fd5fca330dcd6a7… 1.04kB 4 weeks ago /bin/sh -c #(nop) COPY file:0b866ff3fc1ef5b0… 1.96kB 4 weeks ago /bin/sh -c #(nop) COPY file:65504f71f5855ca0… 1.2kB 4 weeks ago /bin/sh -c set -x && addgroup -g 101 -S … 17.8MB 4 weeks ago /bin/sh -c #(nop) ADD file:5d673d25da3a14ce1… 5.57MB Notice that we see the entire history here, and the first entry is from our commit of the running container. The first line listed shows our commit message in the rightmost column. Let's remove this image and check out the other options: ➜ ~ docker rmi mmm Untagged: mmm:latest Deleted: sha256:d717f5e1285ec7a539f1e59908375ef3111f59f176ec0e40ec5835ddc96d9816 Deleted: sha256:6d2a62cfa2e2801b9a9e5ed0a5ccf5e173e621e7fe05b15f32379537464e38ec Option C: Commit Without Pause When you use the commit command, the container will be paused. For our little play container this is unimportant, but you might be doing something like capturing an image of a production system where pausing isn't an option. You can add the –pause=false flag to the commit command, and the image will be created from the container without the pause. ➜ ~ docker commit --pause=false nginx_base wo_pause sha256:d78b9fb9c8a0115dd22ad6d142507d44c6300e90bbc32feb62891092100a0c9a If you don't pause the container, you run the risk of corrupting your data. For example, if the container is in the midst of a write operation, the data being written could be corrupted or come out incomplete. That is why, by default, the container gets paused before the image is created. Let's remove this image and check out the other options: ➜ ~ docker rmi wo_pause Untagged: wo_pause:latest Deleted: sha256:d78b9fb9c8a0115dd22ad6d142507d44c6300e90bbc32feb62891092100a0c9a Deleted: sha256:6d2a62cfa2e2801b9a9e5ed0a5ccf5e173e621e7fe05b15f32379537464e38ec Option D: Change Configuration The last option I want to discuss is the -c or –change flag. This option allows you to set the configuration of the image. You can change any of the following settings of the image during the commit process: CMD ENTRYPOINT ENV EXPOSE LABEL ONBUILD USER VOLUME WORKDIR Nginx's original docker file contains the following settings: CMD ["nginx", "-g", "daemon off;"] ENV NGINX_VERSION 1.15.3 EXPOSE 80 So we will just play with one of those for a moment. The NGINX_VERSION and EXPOSE could cause issues with container startup, so we will mess with the command line (CMD) executed by the container. Nginx allows us to pass the -T command line

argument that will dump its configuration to standard out. Let’s make an image with an alternate CMD value as follows: ➜ ~ docker commit --change='CMD ["nginx", "-T"]' nginx_base conf_dump sha256:0a6cf9c4443e9d0a7722aeaf528ffc6b40622bf991928dfb97bd3db0a3ea6dee Now stop the nginx_base container with this command: ➜ ~ docker stop nginx_base nginx_base And start a new container from the image we just created: ➜ ~ docker run --name dumper -p 80:80 conf_dump The configuration for the nginx process will be dumped to standard out when you execute this command. You can scroll back several pages to find the command we executed. Creating Docker Images: Conclusion The docker commit subcommand is very useful for diagnostic activities and bootstrapping new images from existing containers. As I showed above, there are many helpful options available, too. The Docker CLI has many other power commands. If you like, you can explore some of them here. DataSet is the best-of-breed log analytics solution for dynamic container environments. It is the only solution that provides unmatched performance and scale while optimizing the total cost of ownership. To find out more about working with Docker and DataSet, check out these resources: Installing the DataSet Agent in Docker Configure the DataSet Agent for Docker