**Docker Notes**

**Introduction**

* Docker makes use of containers to run applications in isolated environments. For example, we could have a container for a Node.js app, React.js app, or a MongoDB app.
* A picture containing text, font, screenshot, electric blue

  Description automatically generated
* Suppose we are working on a dev team, and you make a feature that requires Node 17. If we want other people on the team to use this feature, they’d need Node 17 as well. Additionally, they’d need any project dependencies or environment variables. In general, when we want to run our application on another machine, there is a lot of setup required. Additionally, our teammate might have Node 14 and we don’t want their version of Node that they have on their machine to interfere with the version of Node for our feature.
* A container is like a box that contains everything our application needs to run (source code, dependencies, runtime environment, versions)
* A blue background with white text

  Description automatically generated with medium confidence
* This container allows us to run our application in isolation away from any other processes on our computer so it doesn’t matter what other version of Node we have on our computer for example. The only thing our teammate would need to run our code is docker.
* Containers vs Virtual Machines (VMs)
  + VMs have their own full OS which runs on top of the host computer’s OS.
  + Containers share the host computer’s OS kernal, making them more lightweight, require less memory, and quicker.
  + However, there are times when we want to use a VM so it’s not like containers are just always better than VMs

**Installing Docker**

* <https://www.youtube.com/watch?v=8Ev1aXl7TGY&list=PL4cUxeGkcC9hxjeEtdHFNYMtCpjNBm3h7&index=2>

**Images**

* Images are blueprints for containers
* They store the following (they don’t actually have the following inside them)
  + Runtime environment
  + Application code
  + Any dependencies
  + Extra configuration (ex: env variables)
  + Commands
* Images also have their own file system which is independent from the rest of the host computer
* Images are read only. Once you create an image, you cannot change it. If you need to change it, you need to create a new image.

**Containers**

* Containers are runnable instances of blueprints
* A blue screen with white text

  Description automatically generated with low confidence
* When we run the image, it creates a container which is a process that can run our application exactly as outline in the image (meaning it has the correct runtime env, src code, etc)
* Containers are an isolated process, so they run independently of any other process from our computer (meaning whatever runtime environments the host computer has installed won’t affect the container process)
* Now, to share code, we can also share a docker image. Thus, the receiver of this docker images runs it on their own computer and they make a container.
* A blue screen with white text

  Description automatically generated with low confidence

**Parent Images and Docker Hub**

* Images are made up of different layers. Each layer adds something to the image incrementally, so the order of the layers matter
* A screenshot of a computer

  Description automatically generated with medium confidence
* The first layer we start with is typically a parent image. This parent image describes the OS and the runtime environment of the container. For example, we may want Node 17 on a linux distribution.
* The layers after the first layer can be anything else we add to our image such as copying source code to the image, installing dependencies to the image, or specifying any scripts.
* When we work with the parent image, we will work with Docker Hub which an online repository of docker images. It’s like github but for docker instead. Docke Hub contains many pre-made images as the first layer which we can use in our own images. We can search for and download these premade images to our computer.
* Suppose we want to create a project that runs in a node environment.
* Then the initial layer of our image would be a parent node image.
* We can get that image from docker hub by typing “node” into the search bar.
* We should see the first option being the following:
* A screenshot of a chat

  Description automatically generated with low confidence
* To download this image, we run the code in a terminal: docker pull node
* But before we do that, we can scroll down and see the following on dockerhub:
* A screenshot of a computer

  Description automatically generated with medium confidence
* These tags are the different variations of the node images that we can use. They specific node version and an underlying linux distribution.
* Note that it’s good practice to specify a specific image tag since if you just say “docker pull node”, it will always grab the latest version which can change over time. But for the sake of simplicity, we won’t use the tag.
* Inside any terminal we can run “docker pull node”. It doesn’t matter where we are inside our terminal. This is because our node image isn’t going to be downloaded inside our current terminal directory, but rather, docker is going to store it in a special place.
* A screenshot of a computer

  Description automatically generated
* Notice how since we didn’t specify a tag, it said “using default tag: latest” which pulls the latest version
* Now if we go to docker desktop and go to the images tab, we see the following:
* 
* While this node image is meant to be our parent layer, it is still an image nonetheless so we can run it.
* Thus, if we click the run icon, it runs the image to create a container which runs a linux environment with node installed into it. We can ignore any optional settings and just click “run”.
* If we our container runs then immediately terminates, try running “docker run -d -it node” in any terminal instead.

**Dockerfile**

* We learned how to use a parent image, but to add more layers to the image, we make use of a dockerfile.
* A dockerfile is a set of instructions that lists out the different layers and tells docker how to create the image.
* Suppose we have the following “express-api” folder:
* A picture containing text, font, screenshot, number

  Description automatically generated
* A screen shot of a computer program

  Description automatically generated with low confidenceA picture containing text, screenshot, software, font

  Description automatically generated
* Notice we have express and cors as dependencies listed in package.json. Normally, to install these dependencies, we run npm install which installs the project dependencies inside the project folder in a node\_modules folder. Then we run the app.js file via “node app.js” and that would run the app.js file using the version of node installed in the host computer.
* But, we don’t want to use the node version of the host computer, we want to run our application in an isolated container with its own node version.
* Thus, we need to create an image which indicates the version of node and linux distribution we want the container to run, and then additional layers to copy the source code to the image, copy the dependencies to the image, etc.
* To make this image, we create the following dockerfile inside the root folder of our project.
* The name of this file is Dockerfile
* Recall a Dockerfile is a set of instructions. We will make each of these instructions be on a different line inside this file. Generally speaking, each line in a docker file represents a different layer in our image.
* The first instruction/layer of our image will be specifying the parent image. This is the first instruction so it goes to the top of the file.
* 
* “node:17-alpine” is the name of the image we want to pull. We specify we want node version 17 and the alpine distribution of linux. This image will be pulled from the docker hub repository if we haven’t already downloaded it or our own computer if we have.
* On top of the parent image layer, we want to add our own layers.
* The next layer we add is to copy all of our source code (the app.js, package.json, and package-lock.json) into the image.
* 
* The copy keyword lets docker know that we want to copy some files to the image. The first “.” Is a relative path to the directory that docker needs to copy the source files from. Since the src files are in the same folder as the Dockerfile, the path is going to just be “.” which means the current directory. If the source files that we wanted to copy from were in a express-api/src folder, then instead of “.”, we would have “./src”. The second “.” is the path inside the image that I want to copy my source code to. Images have their own folder structure. Thus, the “.” means we copy the files into the root directory of the image that we are creating. Often, we don’t copy into the image’s root directory since it might clash with other files or folders inside the root directory. Thus, we could “/app” instead to copy the files into the image’s root directory’s app folder.
* The next layer we add is to install all of our dependencies. To normally install node dependencies, we would run “npm i”. We do a similar thing in the image.
* We are allowed to specify commands that we want to run in the image as the image gets built. To specify a command, we use the RUN instruction
* A screenshot of a computer program

  Description automatically generated with low confidence
* We specify the npm install command in this example. However, the problem with this command is that it’ll get executed in the image’s root directory but the package.json file is inside the /app folder, so no dependencies will be installed.
* To solve the above problem, we can specify a working directory as shown below
* A screenshot of a computer

  Description automatically generated with medium confidence
* By specifying a working directory, we tell docker that when we run commands on the image in the future (after this workdir /app line), or if we specify paths inside the image, to do it inside the working directory. Now, the COPY command will copy the files to the image’s /app folder and also run the npm install command inside the /app folder.
* Now, we want to run the app.js file by running node app.js. However, we can’t use the RUN command since that means that “node app.js” will be run in build time. An image is not an application, it is a blueprint for a container and the container is what actually runs the application. Thus, it makes sense to use the RUN instruction for installing dependencies since then, our dependencies will be installed and our image will be ready for when we run the container. But we’re not trying to run the “node app.js” command in build time, but rather when we run the application inside a container.
* Instead of RUN, we use the CMD instruction.
* A screenshot of a computer program

  Description automatically generated with low confidence
* We write the command as an array of strings in double quotes. When the container runs, the command inside the array gets executed.
* There is also an EXPOSE instruction
* 
* the EXPOSE keyword in a Dockerfile is used to document which port will be exposed by the container. It serves as a form of metadata to indicate to users of the Docker image that the container expects inbound connections on a specific port. This informs users of the image that they should map the container's port 4000 to a port on the host machine when running the container. This expose instruction is optional, we only need if we are going to be running images as containers using the docker desktop application. This is because the docker desktop will use this port to setup port mapping (more on that later on). If we run containers from the command line, it’s not really needed. But keeping it in just makes it evident to the developer that port 4000 will be exposed.

**Building an Image**

* To build an image, run the command “docker build -t myapp .” inside a terminal in the same directory as the dockerfile.
* The -t stands for tag and allows us to give the image a name. In the above example, we called the image myapp. The . at the end is a relative path to the dockerfile from the terminal’s current directory.
* A screen shot of a computer

  Description automatically generated with medium confidence
* After the build finishes, we don’t see any new file. Rather, docker stores it away in a specific docker folder in our computer.
* But if we go to docker desktop, we can now see the image:
* 

**dockerignore**

* A screenshot of a computer

  Description automatically generated with low confidence
* Within our express-api folder, we usually have a node\_modules folder. However, this would mean that the COPY command would also copy the node\_modules folder to the image as well. This is bad for two reasons. We install the dependencies in the image again as the next line is “RUN npm install”. This reinstallation of the node\_modules could lead to conflicts. The second reason is because it’ll take more time to create the image and copy the files.
* Thus, to indicate that we don’t want to copy the node\_modules folder, we create a .dockerignore file and specify which files/folders we want docker to ignore when it copies to the image.
* A screenshot of a computer

  Description automatically generated

**Starting and Stopping Containers**

* To run a container, go to docker desktop, the images tab, and run a specific container. We can expand the optional settings and change any settings that we want.
* A screenshot of a computer

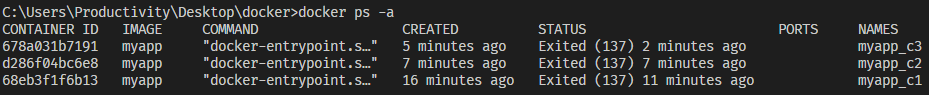
  Description automatically generated with medium confidence
* We gave the container a name, my\_app\_c\_1.
* We also gave it a port, 5000, by which we can reach this container on our machine.
* In our docker image, we exposed the port 4000. We also listen to port 4000 inside our app.js server.
* 
* But the port 4000 is a port exposed by the container. It’s not a port exposed by our computer directly. Thus, if we went to
* The port we specified maps the port on our computer to <http://localhost:4000/> in the browser, it wouldn’t reach our node server.
* To solve this issue, docker allows us to map a local host port number to the port exposed by the container. In our case, we mapped the port 5000 on our machine’s localhost to port 4000 exposed by the container.
* Now, if we go to port 5000 in our browser, it would reach our node server.
* By convention, the port on our computer should map to the same port number exposed by the container.
* Note that we could only map our computer’s localhost to the container’s port number since we exposed a port number in the dockerfile. If we didn’t expose a port in the dockerfile, we’d see the following when we run the image:
* A picture containing text, font, screenshot, graphics

  Description automatically generated
* Thus, our app.js file wouldn’t be able to listen port 4000 since the container doesn’t expose any ports.
* But because we exposed the port 4000 in our dockerfile, app.js knows port 4000 is the only port it can listen to so it listens to port 4000 when starting the server.
* When we run our container, we can see the instance running inside docker desktop’s container tab.
* And if we got to localhost 5000, we can see the following:
* 
* We can stop a container by pressing the stop button. Notice how it doesn’t delete the container, it simply stops it.

**CLI**

* We will use the CLI to handle containers.
* Running containers
* To run an image to create a container, we need either the id of the image or its name.
* To find the specific image, we can list our all the images we currently have by typing in the terminal in any directory: “docker images”
* A screen shot of a computer code

  Description automatically generated with low confidence
* Notice how we can see the name, tag, and id of each image.
* To run an image, run in cmd: “docker run --name myapp\_c1 myapp”
* In this above example, we use the name flag to name the instance of this image as myapp\_c1. We also specify we are running the “myapp” image at the end of the command.
* 
* Now, if we try to go to port 4000, we don’t get anything because of what we described above. The container’s exposed 4000 port doesn’t map to any localhost port.
* Thus, we need to stop the instance and do port mapping.
* Stopping containers
* Notice we can’t stop the instance with ctrl-c. Instead, we need to open a new terminal and run: “docker ps”. Ps stands for processes and this command shows us a list of running containers.
* A black background with white text

  Description automatically generated with low confidence
* Notice we can see the myapp\_c1 container.
* To stop this container, we run: “docker stop container\_name” or “docker stop container\_id”
* 
* Now, if we go to the original terminal that we ran the docker container on, we see that it has stopped.
* Exposing Ports and Mapping Ports
* Now, we want to create a new container that maps the container’s exposed port to a port on the host machine.
* To do so, we run add the -p flag as shown below.
* 
* The port on the left is port is the port on the host machine
* The port on the right is the port exposed by the container
* In our case, we map the host computer’s port 5000 to our container’s exposed port of 4000.
* Now, if we go to local host 5000, we see the server running.
* Since we can expose and map ports via the cli, this is why we previously said the EXPOSE 4000 in the dockerfile is only needed if we are using docker desktop and not the cli.
* Note that we can also add the -d flag which detaches allows us to start the container without it blocking our terminal. An example is shown below:
* 
* Restarting a container
* When we stop a container, we can restart it as well.
* We can run in cmd: “docker ps -a” to list all the containers, not just the running ones.
* 
* To restart one of the stopped containers, we run in cmd: “docker start myapp\_c3”
* 
* Notice we use the “start” keyword and not the “run” keyword since “run” runs an image to create a container while “start” starts up an existing container.
* Notice we specified the name of the container we want to restart. Note that we don’t have to reconfigure any port mapping as that was already created for the container when we ran the image.
* As well docker start doesn’t block the terminal as it runs in detached mode by default

**Layer Caching**

* Docker caching is a mechanism used by Docker to improve the build performance of container images. When you build a Docker image, Docker performs various steps defined in the Dockerfile to create the image. Each step in the Dockerfile generates a new intermediate image layer, which is cached by Docker for subsequent builds.
* Docker caching works by reusing intermediate image layers that have not changed since the previous build. When you make a change in your Dockerfile or add a new instruction, Docker only rebuilds the layers affected by those changes, while reusing the unchanged layers from the cache. This significantly reduces the time required to build the image.
* The caching mechanism is based on the concept of layer reuse. Docker stores each layer as a separate image, and layers are stacked on top of each other to form the final image. When Docker builds an image, it checks if the cache contains a layer with the same instructions and configuration. If it finds a matching layer, it reuses it instead of rebuilding it. This process continues until all the layers are processed.
* To take advantage of caching, it is important to organize your Dockerfile in a way that maximizes layer reuse. For example, placing instructions that change frequently, such as package installations or source code additions, towards the end of the Dockerfile allows Docker to cache the earlier layers that are less likely to change.
* However, caching can also lead to unexpected results if not used carefully. For example, if you have an instruction in your Dockerfile that downloads an external resource, such as a package, Docker will cache the layer containing that resource. If the resource changes, Docker may continue to use the cached layer, resulting in an outdated or incorrect image.
* To control Docker caching, you can use the --no-cache flag during the docker build command to ignore the cache and rebuild all layers from scratch. This can be useful in situations where you want to ensure that you have the latest versions of all dependencies.
* Ex:
  + Suppose we change the src code and we have the following dockerfile
  + A screenshot of a computer program

    Description automatically generated with medium confidence
  + When we build, we get the following:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Notice how the parent image along with the workdir is cached and not rebuilt. This is because the previous time we built this image (before we updated the src code), we were caching each layer of the image while building it. Thus, the image right after the WORKDIR instruction in our previous build is cached. When we perform our current build, we first perform the FROM and WORKDIR instructions. However, since the image from the previous build has been cached and is identical up to the WORKDIR instruction, we don’t reperform those FROM and WORKDIR instructions but simply get the image from the cache. However, since we changed the source code, the image from the previous build in cache is not the same as our current build. Thus, we would have to perform the COPY instruction and all following instructions. That is why we see the word CACHED for the instruction before the COPY in the terminal, but we don’t see the word CACHED for the instructions after and including the COPY instruction.
* Ex:
  + Suppose we have the following dockerfile
  + A screenshot of a computer program

    Description automatically generated with medium confidence
  + Notice we rerun npm install after we copy the source code. This means if we change the source code, we’d have to rerun npm install. Most of the time, changes to our source code don’t change the dependencies. Thus, by running npm install after all source code changes, regardless of if it changed the dependencies or not, it takes up a lot of time. Thus, we can modify the dockerfile to look like the below to have the npm install run before the copying of the source code.
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Notice we swapped the order of the RUN npm install and COPY . . instructions so the node\_modules aren’t rerun every time the source code changes.
  + Notice we also added a COPY package.json . instruction before the RUN npm install. If we didn’t have this instruction, the npm install wouldn’t work as there would be no package.json file and the package.json lists the dependencies. As well, if we were to add a dependency when changing our source code, it would change the package.json file too. Thus, updates to the dependencies would be captured in package.json and npm install would run since the package.json in the cache isn’t the same as the new package.json

**Deleting Containers and Images**

* To delete an image, run in cmd: “docker image rm myimagename”
* In the above example, we deleted the image with a name of “myapp myimagename”
* Note that we can also use the id of the image instead of its name
* A picture containing screenshot, text, font, line

  Description automatically generated
* Note that we need to delete all containers (regardless of if they are running or not) for an image before we can delete that image.
* To delete a container, run in cmd: “docker container rm mycontainername”
* A picture containing text, screenshot, font

  Description automatically generated
* To delete multiple containers, we do the following:
* 
* To delete a container when it stops running, use the --rm flag which stands for remove as shown below:
* A screen shot of a computer

  Description automatically generated with medium confidence
* Suppose we want to an image even though it has container instances of it. To do so, we add the -f flag.
* A picture containing text, screenshot, font, line

  Description automatically generated
* To delete all containers, images, and volumes (more on that later), we can run “docker system prune -a”

**Managing Images**

* Recall that for the parent image, we could specify different versions.
* The different versions were denoted via tags on dockerhub
* A screenshot of a computer

  Description automatically generated with medium confidence
* To create a tag, we add a colon after the image name and then a tag after the colon as shown below
* A screen shot of a computer

  Description automatically generated with medium confidence
* In the above we create a myapp image with a tag of v1
* We then run this image by specifying myapp:v1 in the docker run command

**Volumes**

* Currently, if we want to make changes to our source code, we have to build a new image, then run the image in a container.
* If we are frequently making changes, this becomes very tedious
* A solution to this problem is by using volumes
* Volumes are a feature in docker that allow us to specify folders on the host computer and make them available to containers. We do so by mapping folders from the host computer to the container.
* Thus, if we changed something in those folders on our computer, those changes would be reflected in the folders we mapped to in the container. These changes would be applied automatically and without having to build a new image.
* Note that volumes don’t actually change the images as images are read only. Thus, if we want to share our image with others, then you’d have to build the image using docker build again. Nonetheless, volumes are useful for local development.
* Suppose we have the following files:
* A screenshot of a computer

  Description automatically generated with medium confidence
* Below is the npm run dev script in the package.json:
* A picture containing text, screenshot, font

  Description automatically generated
* Below is the dockerfile
* A screenshot of a computer program

  Description automatically generated with medium confidence
* Suppose we create this image using docker build a shown below:
* A picture containing text, screenshot, font

  Description automatically generated
* We can run a container as shown below and map our local project folder to the container app folder.
* To map these folders, we use the volume flag using -v before the image name. First, we specify the absolute path on our host computer to our project folder. Then we add a colon and then a path to the folder we want to map in the container.
* A picture containing text, screenshot, font

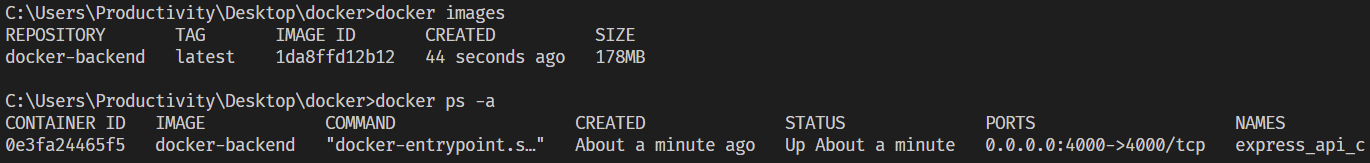
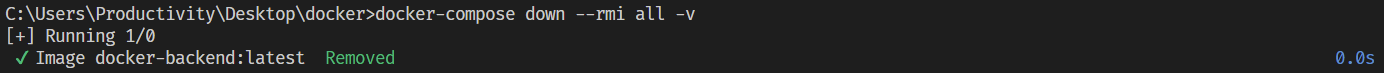
  Description automatically generated
* In this example, we map our host computer’s \express-api folder to the container’s /app folder.
* Now if we go to local host 4000 and update the src files, we see the changes.
* However, one problem with the above is that if we delete the node\_modules folder in our host machine’s express-api folder, it would remove the node\_moduels folder inside the container’s /app folder as well since the two folders are synced using volumes. This is a problem since the container needs the node\_modules folder to work.
* Thus, we need to make it so that the host machine’s express-api folder and container’s /app folder are synced, but deleting the host machine’s express-api/node\_modules folder doesn’t delete the container’s /app/node\_modules folder.
* The way we solve this is to use another volume called an anonymous volume. This new volume would map the container’s node\_modules folder to somewhere else on our host computer. Docker takes care of where this other folder on our host computer is.
* We can create this anonymous volume as shown below:
* A screen shot of a computer

  Description automatically generated with medium confidence
* Note that the -v /app/node\_modules flag is a volume that specifies the location of our container’s app/node\_modules folder. Then, this folder is mapped to some folder on our host machine managed by docker. The contents of that folder will persist even when the container stops. Thus, the next time the same container starts again, the container’s node\_modules folder is still mapped to that same folder on our host machine managed by docker.
* The effect of this new volume is that it overrides the previous volume but only for the node\_modules folder. The reason it does that is because the anonymous volume’s path in the container is more specific than the path of the previous volume.

**Docker Compose**

* We’ve seen how to run an image while matching ports and volumes, but that is a really long script.
* Additionally, we might have multiple projects and we want to containerize each, then run all the containers simultaneously.
* An alternative to writing a long script is to use docker compose.
* Docker compose allows us to make a docker compose file that indicates all the container configurations for our project such as port mappings, volumes, container names, etc.
* An example compose file is shown below:
* A screen shot of a computer

  Description automatically generated with medium confidence
* Line 1 indicates the “version” keyword of docker being used. We can check our the version by running in cmd: docker --version
* Line 2 uses the “services” keyword which can take on multiple values. Services are the projects that we want images and containers for. Currently, we only have the express-api project
* We can name each service however we want. In the above, we created a service and called it “backend”. Note the indent.
* Then we configure this “backend” service.
* We make use of the “build” keyword which takes a relative path to the folder (relative to the docker compose file) where the dockerfile for that project is. This is because docker compose still uses the dockerfile that we previously created. In our case, the docker file’s path relative to the docker compose file is “./express-api”
* We then use the “container\_name” keyword which tells docker compose what to call our container when we run this backend image. In our case, we call it “express\_api\_c”
* We then use the “ports” keyword to indicate how we want to map the host computer’s ports to the container’s port when we run the image to make a container. Note that we can map as many ports as we want so it’s a list of port mappings
* We then use the “volumes” keyword to indicate the volumes we want to create when we run the image to make the container. Note that we create as many volumes as we want so it’s a list of volumes. Note that for specifying the host machine’s folder that we want to map to the container’s folder, we don’t have to use an absolute path. Rather, we just use a path relative to the docker-compose file.
* Running docker compose
* Running a docker compose file does two things:
  + Look at the build property for each service and it’s going to find the dockerfile for each of those services and use them to build images for each service.
  + Automatically run those images to create a container for each image. These containers will have the ports, names, volumes, etc as specified in the docker compose file.
* To run the docker compose file, open cmd and navigate to the same folder as the docker-compose file. Then run in cmd: “docker-compose up”
* A screen shot of a computer

  Description automatically generated with medium confidence
* Now if we check our docker images and containers, we see:
* 
* To stop the containers and remove it when they are done, we run: “docker-compose down”.
* 
* Note that the images and volumes would still remain. If we wanted to remove the images and volumes as well, we could tag on the --rmi all -v flag.
* 

**Dockerizing React**

* Suppose we have the following react app:
* A screenshot of a computer program

  Description automatically generated with low confidence
* Notice there is a .dockerignore file which only contains the node\_modules
* Suppose we have the following Dockerfile:
* A screenshot of a computer program

  Description automatically generated with medium confidence
* Note that EXPOSE 3000 is once again only needed if we are using docker desktop to build and run images
* Now we could have the following docker-compose file:
* A screen shot of a computer

  Description automatically generated with medium confidence
* Note that without the environment and watchpack\_polling=true configuration, changes to the react src code won’t be reflected in the container even though we made a volume for it (since windows is weird)
* "stdin\_open": This option allows you to keep the standard input of the container open, enabling you to interact with the running container's process through the command line. By default, the standard input is closed, which means you cannot enter any input once the container is running. Setting "stdin\_open" to true allows you to attach to the container's stdin, which is useful for scenarios where you want to provide input to the running process.
* "tty": This option allocates a pseudo-TTY (terminal) for the container. It is typically used in conjunction with "stdin\_open" to enable an interactive terminal session within the container. When "tty" is set to true, Docker allocates a TTY device and connects it to the container's stdin. This option is useful for running command-line applications that expect a terminal-like interface.
* Together, these “stdin\_open” and “tty” options allow you to interact with the container's process through the command line as if you were running commands directly in a terminal. They are commonly used when you need to execute interactive commands or debug a containerized application.

**DockerHub**

* To share our images in dockerhub, we first create an account.
* When we log in, click on create a repository, give the repository any name you want, and make it public.
* Ex:
* A screenshot of a computer

  Description automatically generated with medium confidence
* To push an image, we must first create an image locally and the name of this image must be the same as the repository.
* 
* Then we login to docker via the terminal by running in cmd: “docker login”
* Then, to push our image up to the repository, we run:
* 
* Now, if we go the dockerhub and refresh the repository, we should see the following:
* A screenshot of a computer

  Description automatically generated with medium confidence
* Now, to pull down this image, we run in cmd:
* A picture containing text, screenshot, font, white

  Description automatically generated
* 
* Now that we have this image, we can build it and run it to create containers