# Docker Commands Sheet

|  |  |
| --- | --- |
| docker rm <<node-app>> -f | Force-kill currently running Docker container |
| docker build -I <<node-app-image>> . | Build Docker image |
| docker run -p <<incoming traffic port>>:<<node container port>> -d --name <<name of container>> <<name>> | Creates a writeable container layer over the specified image, and then starts it using the specified command |
| docker exec -it <<node-app>> bash | Log into Docker container |
| docker ps -a | Show all Docker containers |
| docker volume ls | Get list of Docker volumes |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d --build | Start up the Docker container using the supplied files |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --build --no-deps node-app | Same as above, but with a specific service/container name, and make sure no dependencies are built either (so only builds one) |
| docker-compose -f docker.compose.yml -f docker.compose.dev.yml down -v | Shut down the Docker container using the supplied files |

**Multiple Containers**

|  |  |
| --- | --- |
| docker exec -it node-docker\_mongo\_1 mongo -u "alan" -p "mypassword" | Log into Docker Mongo container directly (without using bash) |
| docker network ls | Look at all Docker networks |
| docker inspect <<container name>> | Get more information about a container |
| docker network inspect <<container name>> | Get more information about a container's network |

**Production Environment**

|  |  |
| --- | --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml build | Build an image |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml push | Push an image to Docker Hub |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml push node-app | Push a specific image to Docker Hub |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml pull | Pull the image |

**Docker Swarm**

|  |  |
| --- | --- |
| docker stack deploy -c docker-compose.yml -c docker.compose.prod.yml <<app name>> | Deploy the application using Docker Swarm |
| docker node ls | List out all of the nodes within Docker Swarm |
| docker stack ls | List all of the stacks |
| docker stack services <<app name>> | List all the services within a stack |
| docker service ls | List all of the services across all stacks |
| docker stack ps <<app name>> | List alll of the tasks for the stack |

# Mongo Commands Sheet

|  |  |
| --- | --- |
| **Command** | **Result** |
| db | Shows us what database we are connected to (mongo creates a test database so we have some database to log into) |
| use mydb | Creates a new database and switches to it |
| show dbs | Shows all databases   * mydb is not listed on here because mongodb won’t list databases until there is a document or entry within it |
| db.books.insert({"name": "harry potter"}) | Inserts a document into the books collection |
| db.books.find() | List all documents within the books collection |
| exit | Quit mongo |

# Installing Docker

Using <https://blog.devgenius.io/installing-docker-onwindows-10-home-edition-2e7c1b79d76d> for basis

## Install WSL

* Open Command Prompt in Administrator mode
* Install WSL by running the following command:

|  |
| --- |
| dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart |

* Now update to WSL2 by running the following command:

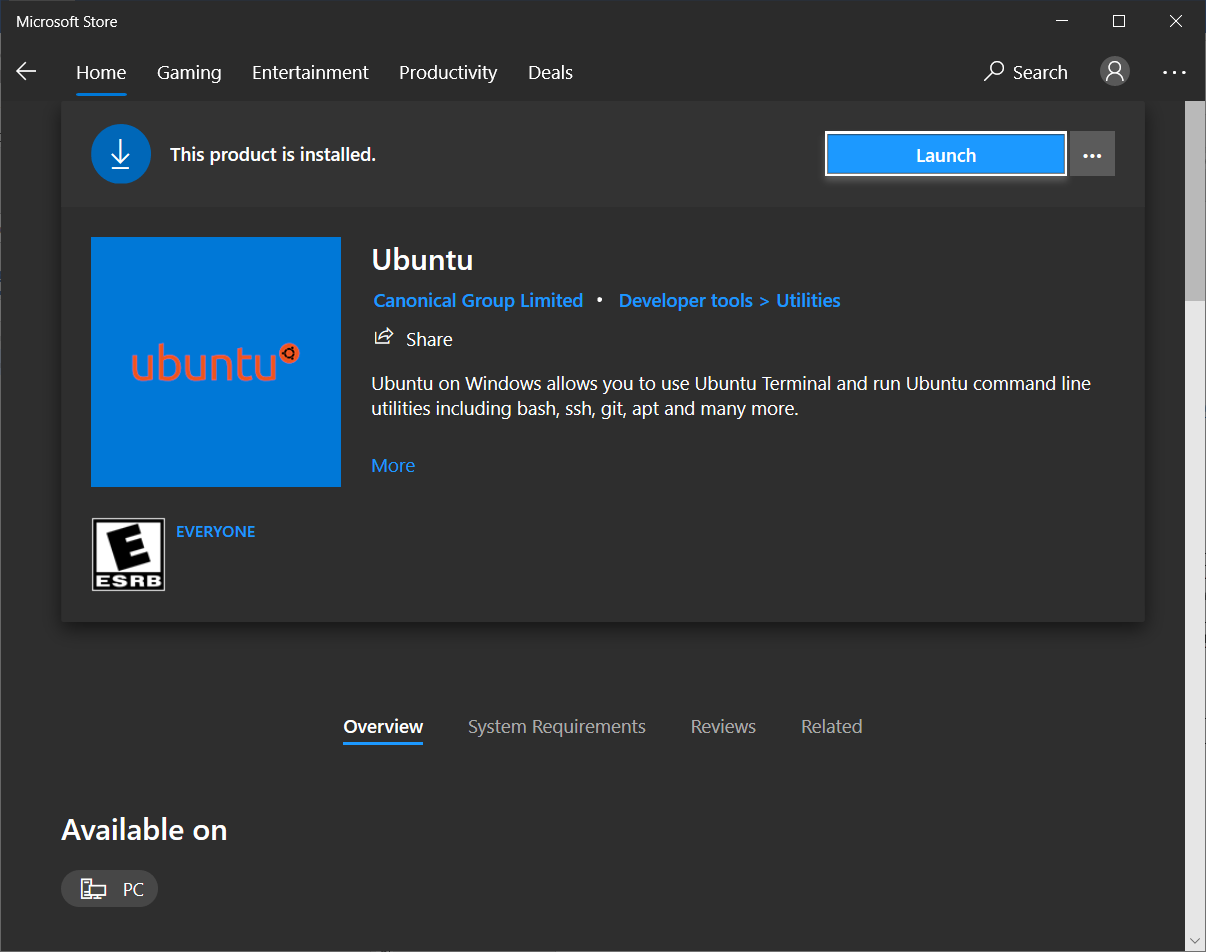
|  |
| --- |
| dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart |

* Restart your machine
* Re-open Command Prompt in Administrator mode and run the following command:

|  |
| --- |
| wsl --set-default-version 2 |

## Install Linux distro

* Open Microsoft Store
* Choose your favourite one; if you don’t have one then just install Ubuntu

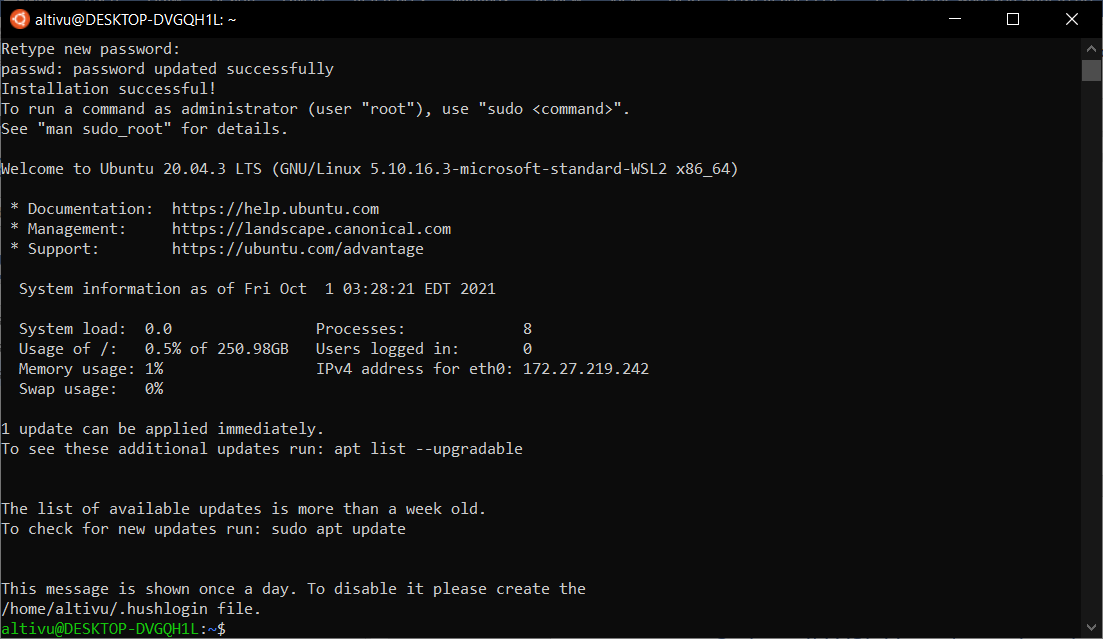


## Install WSL Linux kernel update

* Download and run the executable here: <https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi>
  + This will download the WSL2 Linux kernel update

## Run Ubuntu

* Run Ubuntu; on first go, it will prompt to create a username and password, so do so



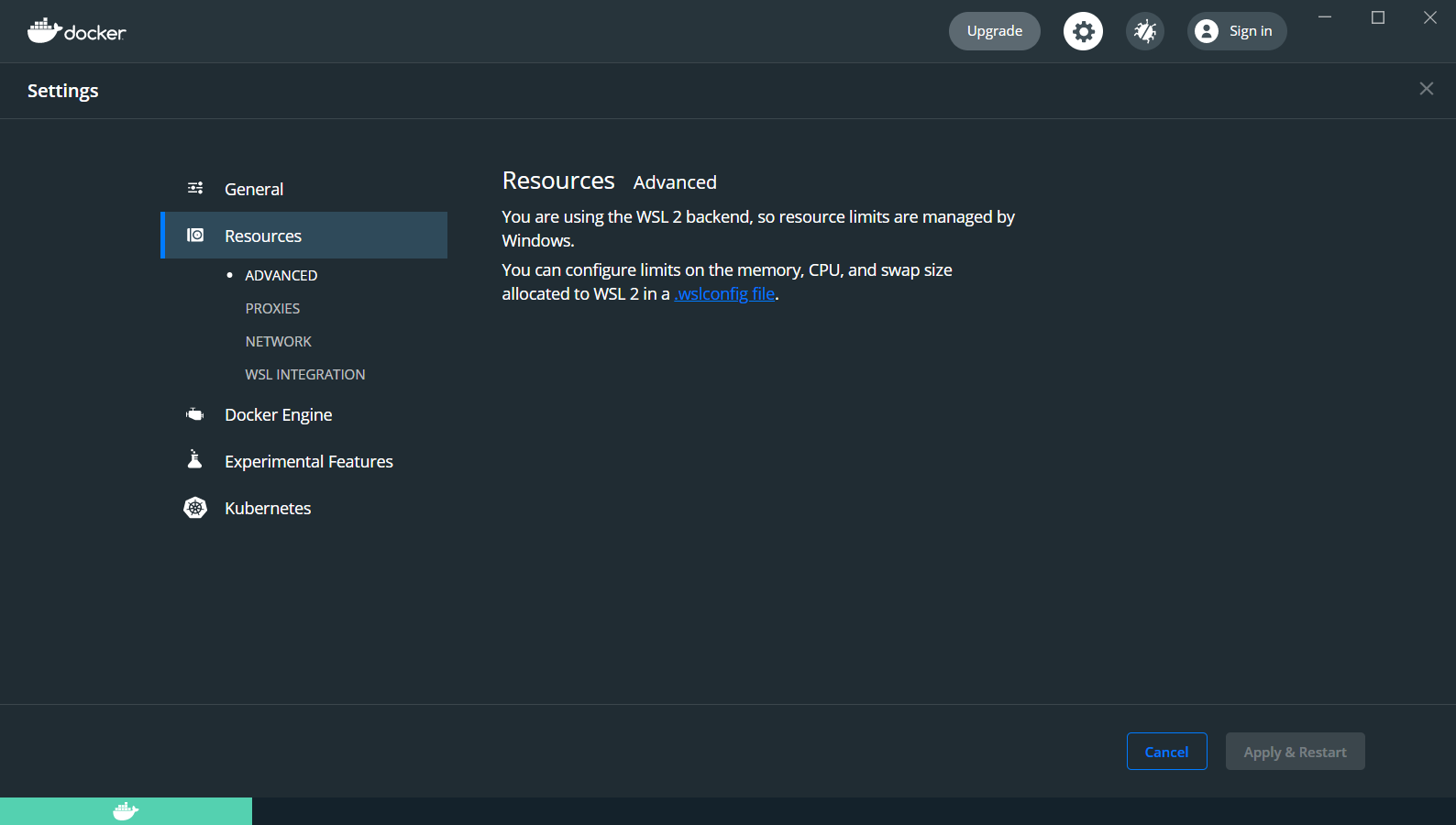
## Install Windows Terminal (optional)

* In the Microsoft Store, install Windows Terminal

<https://codefellows.github.io/setup-guide/windows/>

## Install Docker Desktop

* The above steps should be completed before installing Docker Desktop, otherwise you may not see the Settings > Resources > WSL INTEGRATION component available
* Install Docker Desktop at <https://www.docker.com/products/docker-desktop>
  + Follow all instructions as standard
* You will know things are going okay if the Settings > Resources window looks something like this:



Learn Docker - DevOps with Node.js & Express (hosted on freeCodeCamp): <https://www.youtube.com/watch?v=9zUHg7xjIqQ>

# Part 01: Building the app

## 0:00:14 Intro & demo express app

* Set up an express app
  + Create a node-docker folder and navigate into it
  + Create a package.json file by running the following command:

|  |
| --- |
| npm init -y |

* Start installing the required dependencies:

|  |
| --- |
| npm install express |

* Create an index.js file with the following content:

|  |
| --- |
| const express = require("express");  const app = express();  app.get("/", (req, res) => {    res.send("<h2>Hi there!</h2>");  });  const port = process.env.PORT || 3000;  app.listen(port, () => console.log(`Listening on port ${port}`)); |

* Start the app by running the following:

|  |
| --- |
| node index.js |

* + You should see the app open in the browser with the message “Hi there!”

## 0:04:18 Custom Images with Dockerfile

* Go to hub.docker.com, log in, and search for the node docker image
  + <https://hub.docker.com/_/node>
* To create a custom image, create a Dockerfile file (no extensions)
* First command is to always specify a base image/known image of some sort that Docker has access to:

|  |
| --- |
| FROM node |

* + To specify a specific version, use node:<version>
* The next command is technically optional but still recommended, which is the WORKDIR command:

|  |
| --- |
| WORKDIR /app |

* + What this does is it sets the working directory of the container to be the /app directory in the container
  + Setting a work directory because any time you run a command, when you set the work directory, it will run that command from the directory
  + Also the directory where if you copy any files to the container, it will by default send it to that directory
* Next step is to copy the package.json file, which contains all the dependencies, into the docker image:

|  |
| --- |
| COPY package.json . |

* + Copy it to the current directory (which in this case will be the /app directory due to the previous line
* Now the next step is to actually install the dependencies:

|  |
| --- |
| RUN npm install |

* Now copy the rest of the files/source code into the docker image:

|  |
| --- |
| COPY . ./ |

* + The reason the “COPY package.json .” was split from this line, even though it would technically already copy the package.json file, is that it is a little bit of an optimization

## 0:10:34 Docker image layers & caching

* When you create an image from a docker file, it takes each one of these steps and it treats it as a layer of the image
* After each layer, Docker caches the results of each step/line, if the image is rebuilt, if steps haven’t been changed, then the cached results allow the image to be built much faster
* By splitting this up into two different steps, each of the layers are cached, and instances where the code changes but package.json does not, it will be faster jumping to a later step
* If a previous step is changed, then all subsequent steps will also need to be rebuilt
* Next step is to expose the port 3000:

|  |
| --- |
| EXPOSE 3000 |

* Finally, when the container is started, we want to tell it what command to run
  + Since it is a node application and the entry point into the application is index.js:

|  |
| --- |
| CMD ["node", "index.js"] |

* + This last step is specificially at runtime
* Now build the docker image by running the following in the command prompt; you need to also specify the path (context) to the docker file:

|  |
| --- |
| docker build . |

* When it is done, you can check the image by running:

|  |
| --- |
| docker image ls |

* + You can see the new image created without a name and image that it pulled from docker hub
* You can delete a docker image by its image id with the following:

|  |
| --- |
| docker image rm <<image id>> |

* To build a docker image with an actual name, use the following:

|  |
| --- |
| docker build -t <<name>> . |
| docker build -t node-app-image . |

* To run the image:

|  |
| --- |
| docker run -d --name <<name of container>> <<name>> |
| docker run -d --name node-app node-app-image |

* **The -d flag stands for detached, so you can access the image outside of the CLI**
* The --name tag is for giving a name to the docker container

## 0:20:26 Docker networking opening ports

* The “EXPOSE 3000” line does not actually do anything and is more for documentation purposes; it is just that when the docker file is shared with someone else, they will know that the image expects port 3000 to be opened for it to work
* The thing about Docker containers cannot be talked to by outside devices by default
  + To make the container available to “the outside world” (which includes the internet, but also a local host machine) is to set on the host machine, if it receives traffic on a specific port, to forward the traffic to the Docker container
* You can delete a running container by its name as well:

|  |
| --- |
| docker rm <<container name>> -f |
| docker rm node-app -f |

* + The -f flag is for force, which allows you to delete running containers; normally you would need to stop the container before deleting it
* To run a container with a port flag:

|  |
| --- |
| docker run -p <<incoming traffic port>>:<<node container port>> -d --name <<name of container>> <<name>> |
| docker run -p 3000:3000 -d --name node-app node-app-image |

* The port number to the right is the port we are going to send traffic to on our container
  + Our container is listening on port 3000, so set that right number to 3000
* The port number to the left represents traffic that is going to be coming in from the outside world (such as another device on your network, or your local host/Windows machine); if these send traffic to local host port 3000, we are taking traffic coming in on port 3000, and sending it to the number on the right
* Run the following to look at current docker containers:

|  |
| --- |
| docker ps |

* The port will say 0.0.0.0.3000 -> 3000/tcp; what this is saying is that any traffic destined to your host machine on port 3000 will be forwarded to port 3000 on the container

## 0:26:36 Dockerignore file

* Log into Docker container using the following:

|  |
| --- |
| docker exec -it node-app bash |

* + -it stands for interactive
  + bash allows us to take a look at the file system of the container
  + This will drop us into the /app directory by default, as it was set as the working directory
  + We can exit this screen by typing exit
* The Dockerfile is there to create an image, but it is not required in the Docker container
* Ultimately, we want to move away from developing on a local machine; moving forward, we won’t even have a node\_modules folder on our local machine
* We need to find a way to make it so that Docker doesn’t copy files that we don’t want copied over, such as Dockerfile, node\_module, git configs, etc.
* This can be done by creating a .dockerignore file
* Create a .dockerignore file, and in this file, list out every file and folder that we don’t want copied over:

|  |
| --- |
| node\_modules  Dockerfile  .dockerignore  .git  .gitignore |

* Build the docker image, and run the container

## 0:31:46 Syncing source code with bind mounts

* If we make changes to the code after a build, the code in our image will now have a stale version
  + You can confirm this by logging into the Docker container and viewing the file contents to see that the changes are not present in the file (such as through the cat command)
* Normally to push any new changes, you will have to remove the Docker container and rebuild/redeploy it
* To have persistent data in the Docker containers, we can take advantage of something known as volumes; a specific type of volume called a “bind mount”, which allows us to sync a folder in our local machine to a folder within our Docker container
  + We can take all the files and sync them to the /app directory of our container, so that we don’t have to continuously rebuild and redeploy every time we make changes
* First, delete the container
* Now run Docker but with a new tag; the below command includes the entire line:

|  |
| --- |
| docker run **-v <<path to folder on local machine>>:<<path to folder on container>>** -p <<incoming traffic port>>:<<node container port>> -d –name <<name of container>> <<name>> |
| docker run **-v C:\Users\AltiV\Documents\Programming\Docker\node-docker\:/app** -p 3000:3000 -d –name node-app node-app-image |

* + The -v tag stands for “volume”
  + The <<path to folder on local machine>> requires an absolute path; it does not simply accept “.” F or example
  + The above line is starting to get very lengthy, so we can make use of variables to make it more straightforward:
    - <<<<path to folder on local machine>> can be replaced one of the following, assuming you are in the proper current working directory:

|  |  |
| --- | --- |
| CMD | %cd% |
| Powershell | ${pwd} |
| Mac/Linux | $(pwd) |

* Any time we make changes to code in a node or express application, we have to restart the node process
  + To handle this, we can install nodemon so that when any code changes are made, it will automatically restart the process so that the changes are updated in real-time
* Run “npm install nodemon –save-dev” to install nodemon
* Set up the following scripts block in package.json to make use of nodemon in a dev environment:

|  |
| --- |
| “scripts”: {      "start": "node index.js",      "dev": "nodemon -L index.js",    }, |

* + **If you run into an issue concerning nodemon not properly restarting, you may need to pass the -L flag as included above**
* Re-build the image (since changes were made to the code)
  + Note that it will take longer this time, as the package.json file changed, since that Steps 3 to 5 have to be re-run
* In the Dockerfile, since we are now using nodemon, change the last line from

|  |
| --- |
| CMD [“node”, “index.js”] |

to

|  |
| --- |
| CMD [“npm”, “run”, “dev”] |

* Now, assuming the volume is being used when running the Docker container, nodemon should now handle code changes properly, and local changes should be reflected in the container

## 0:45:30 Anonymous Volumes hack

* As a test, delete the node\_modules folder on the local side and redeploy the container, which will break the application
* If you do a docker ps, you will notice the container is not showing up
* If you do a docker ps -a, it will show you all containers, and you should see the container you are working with which has stopped
* To view the logs for the container, run the following:

|  |
| --- |
| docker logs <<name of Docker container>> |
| docker logs node-app |

* If you look into the logs, you will see an error stating that nodemon was not found
* The issue that is occuring is that when the docker run command was used, the bind mount syncs the folder with the /app folder, which means that when the node\_modules folder is deleted on the local side, it is also deleted in the container
* To get around this issue, create a new volume (known as an anonymous volume)
  + Delete the broken container with docker rm node-app -f
  + Run the Docker run command again, but specify another volume:

|  |
| --- |
| docker run -v <<path to folder on local machine>>:<<path to folder on container>> -v /app/node\_modules -p <<incoming traffic port>>:<<node container port>> -d --name <<name of container>> <<name>> |
| docker run -v ${pwd}:/app -v /app/node\_modules -p 3000:3000 -d --name node-app node-app-image |

* + This is a hack that you can do to prevent the bind mount from overwriting the /app/node\_modules folder, because all volumes in Docker containers are based off of specificity
  + Even though this bind mount should sync with the /app directory, we can see that we have another volume that references the /app/node\_modules folder; we can see that since this is a longer path, it is more “specific”, which prevents the bind mount from deleting the node\_modules folder
  + The extra “-v /app/node\_modules” line is basically saying “do not touch this folder since it is a more specific/longer path, such that it will overwrite the bind mount
* The “COPY . ./” line in the Dockerfile, despite having a bind mount now, is still required because the bind mount is just for the development process; for production, there will be no bind mount

## 0:51:58 Read-Only Bind Mounts

* The binding is a two-way street; if a new file is created in the container itself, it will also show up in the local machine
  + Do you think that there will ever be an instance where the Docker container undergoes changes that should then be reflected in the local machine? Probably not
* In this case, we can take the bind mount created and make it read-only, such that the Docker container can read any of the files, but cannot touch or create any files
  + Kill the container with docker rm node-app -f
  + To make the container read-only, specify a colon at the end of the /app:

|  |
| --- |
| docker run -v <<path to folder on local machine>>:<<path to folder on container>>**:ro** -v /app/node\_modules -p <<incoming traffic port>>:<<node container port>> -d --name <<name of container>> <<name>> |
| docker run -v ${pwd}:/app**:ro** -v /app/node\_modules -p 3000:3000 -d --name node-app node-app-image |

* Now if the container is run and a file is attempted to be created (such as through the touch command), you will receive an error stating that it is a Read-only file system

## 0:54:58 Environment variables

* After the COPY command in the Dockerfile, add an environment variable PORT with a default value of 3000, and change the EXPOSE line to reference the variable:

|  |
| --- |
| FROM node  WORKDIR /app  COPY package.json .  RUN npm install  COPY . ./  **ENV PORT 3000**  **EXPOSE $PORT**  CMD ["npm", "run", "dev"] |

* Kill and rebuild the image with the usual rm and build commands
* Now with the run variable, you can now pass in an environment variable by doing –env or -e:

|  |
| --- |
| docker run -v <<path to folder on local machine>>:<<path to folder on container>>:ro -v /app/node\_modules -e PORT=4000 -p <<incoming traffic port>>:<<node container port>> -d --name <<name of container>> <<name>> |
| docker run -v ${pwd}:/app:ro -v /app/node\_modules -e PORT=4000 -p 3000:4000 -d --name node-app node-app-image |

* + If the port is set to 4000 as in this example, since the express application is listening on port 4000, we have to change the port that we are sending traffic to; we were sending traffic to the container on port 3000, but our express app is listening on port 4000, so we have to change that second number to 4000 or else the application will break
  + We don’t have to change the port that we have to hit on the local machine
* To check that the environment variables have been properly set, you can run the “docker exec -it node-bash-app” command, and run the following:

|  |
| --- |
| printenv |

* + You should see in this scenario that PORT=4000

## 0:59:16 Loading environment variables from file

* If you have multiple environment variables to deal with, it can be cumbersome to add them all to the Dockerfile line by line
  + We can instead create a new file to hold these environment variables; the standard is a .env file
* To load the environment variables from the file instead of setting them from the command line, change the docker run command to something like such:

|  |
| --- |
| docker run -v <<path to folder on local machine>>:<<path to folder on container>>:ro -v /app/node\_modules **--env-file <<location of file>>** -p <<incoming traffic port>>:<<node container port>> -d --name <<name of container>> <<name>> |
| docker run -v ${pwd}:/app:ro -v /app/node\_modules **--env-file ./.env** -p 3000:4000 -d --name node-app node-app-image |

## 1:01:31 Deleting stale volumes

* You can get a list of Docker volumes through the following command:

|  |
| --- |
| docker volume ls |

* The reason there are so many volumes is that every time a container is deleted, the annonymous volume holding the node\_modules is preserved, which we don’t actually need
* To manually delete the volume, you can run one of the following commands:

|  |
| --- |
| docker volume rm <<volume name>> |
| docker volume prune |

* Alternatively, you can alter the “docker rm” command to also remove the volume that is associated with the container as such:

|  |
| --- |
| docker rm node-app -fv |

## 1:04:01 Docker Compose

* When it comes to creating our container, we have set up a nice quick workflow for developing a node/express application in the container, however, the command to run the docker container is (still) kind of long, and we usually don’t want to have to re-run the command every time we want to get the container running
  + As we continue developing, we will probably have more than one container in a full-blown application, such as a container for our database, an elastic search container, a container for redis, and so on, which will each require a command equally long
* To alleivate this issue, we can use something known as “docker compose”, where we can create a file that has all the steps and configuration settings we want for each docker container
  + For example, we can say that we want to create a node container using the image that we created, create a volume for the bind mount, create an anonymous volume, pass in environment files, and open up ports, and so on
  + We can then run one command to handle all of that
* Create a new file called “docker-compse.yml”
* The first thing to do with this file is to specify which version to be used
  + You can go to <https://docs.docker.com/compose/compose-file/> to see the different available versions for docker compose
  + For now we can just default to version 3:

|  |
| --- |
| version: "3" |

* The next thing we can to do is specify all of the containers that we want to create; within our docker compose file, each container is referred to as a service:

|  |
| --- |
| version: "3"  services:    node-app:      build: .      ports:        - “3000:3000”      volumes:        - ./:/app:ro        - /app/node\_modules      environment:        - PORT=3000      # env\_file:      #   - ./.env |

* + The “node-app” is designated as the name of the container, and stuff nested underneath is specific configuration settings for that container
  + Spacing matters; make sure nested blocks are handled by one tab each
  + If you have more than one environment variable, then consider using the env\_file block to reference a file; since only one environment variable is being used here, we will just stick with the environment block
* Now in the command prompt, run the following:

|  |
| --- |
| docker-compse up -d |

* + Thils builds the image and also starts the container; if you do a “docker ps”, you should see the container running with a name of <<project-name>>\_<<name of service>>\_1 (i.e., node-docker\_node-app\_1)
* To bring this down, run the following:

|  |
| --- |
| docker-compose down -v |

* + Technically when you run docker compose, it technically creates a new separate network for all of your services
  + You can confirm the container is no longer running by checking “docker ps” again
* If you run the “docker-compose up” command again, it will actually skip the entire build process and create the network and start the container, as it looks for an image and sees if it already exists
  + Even if we make a change, such as changing the default port from 3000 to 4000 (which technically changes the image), which we assume would make it require to rebuild the image, it actually did not rebuild the image; it is a stale image now (which is bad)
* To force a rebuild image, add a –build flag

|  |
| --- |
| docker-compose up -d –build |

* Right now when we run “docker compose”, everything is with respect to our development environment, because our docker compose creates a bind mount (which we would never want in production deployment due to it not needing to sync with everything), and our production deployment may use different variables (such as with the port), and it will typically run a different script (not using nodemon)

## 1:21:36 Development vs Production configs

* You can technically create multiple Dockerfiles for development and production to differentiate between the two
  + Some people recommend not using the “npm run” command, as it is another layer between node and the container (so use “node index.js” instead of “npm run start”, but this is somewhat subjective)
* We can also create two different Docker compose files to handle this
* This section will deal with using one Dockerfile only, but two other docker-compose files
  + Rename the docker-compose.yml file to “docker-compose.backup.yml” for reference
  + Create three docker-compose files:
    - docker-compose.dev.yml
    - docker-compose.prod.yml
    - docker-compose.yml
  + The docker-compose.yml will have any configurations that are shared between both environments
* This is what we will have in our docker-compose.yml which is shared:

|  |
| --- |
| version: "3"  services:    node-app:      build: .      ports:        - "3000:3000"      environment:        - PORT=3000 |

* + For now, we are sharing ports between both environments
* Now within our dev and prod files, we can go into our services, create a node-app section, and overwrite anything that we wanted to (such as the ports, or adding extra configuration sections)
* Dev:

|  |
| --- |
| version: "3"  services:    node-app:      volumes:        - ./:/app:ro        - /app/node\_modules      environment:        - NODE\_ENV=development      command: npm run dev |

* + The command line is to overwrite what is in the Dockerfile; for this, the dev runs nodemon
* In the Dockerfile, change the last line from “CMD ["npm", "run", "dev"]” to account for different environments:

|  |
| --- |
| CMD ["node", "index.js"] |

* Prod:

|  |
| --- |
| version: "3"  services:    node-app:      environment:        - NODE\_ENV=production      command: node index.js |

* To run a specific docker-compose file (in this case it is the dev file), run the following command (order matters):

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d --build |

* + The first file is the base file, with the subsequent file overwriting anything that has similar lines
  + The --build flag is to build the image before creating the container
* Shut down the container:

|  |
| --- |
| docker-compose -f docker.compose.yml -f docker.compose.dev.yml down -v |

* Now try composing with the production environment:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --build |

* You will notice that this time, any changes done on the local side will **not** reflect on the container (i.e., localhost:3000 in this scenario), which is expected, as there is no bind mount
* Add the docker-compse files to .dockerignore by adding the following:

|  |
| --- |
| docker-compose\* |

* Right now, nodemon is found in the production Docker container, which we don’t actually want because nodemon is a development dependency; we should remove this so it does not take up space (along with any future dev dependencies)
  + To solve this manually, you can add “--only=production” after the “RUN npm install” line in Dockerfile, but we want to set it up to be intelligent enough to determine whether or not we are running in development or production
  + To do this, we will need to write an embedded bash script
* The full Dockerfile will look like this after adding the bash script:

|  |
| --- |
| FROM node  WORKDIR /app  COPY package.json .  RUN npm install  **ARG NODE\_ENV**  **RUN if [ "$NODE\_ENV" = "development" ]; \**  **then npm install; \**  **else npm install --only-production; \**  **fi**  COPY . ./  ENV PORT 3000  EXPOSE $PORT  CMD ["node", "index.js"] |

* + The NODE\_ENV variable will have to be added to the docker-compose file; we will overwrite something in our base file
* In the docker-compse.dev.yml file, edit the “build:” code block to add more variables:

|  |
| --- |
| version: "3"  services:    node-app:      build:        context: .        args:          NODE\_ENV: development      volumes:        - ./:/app:ro        - /app/node\_modules      environment:        - NODE\_ENV=development      command: npm run dev |

* Do the same for production, changing the NODE\_ENV to “production”
* Bring the container down with the usual “docker-compose -f docker-compse.yml -f docker.compose.prod.yml down -v”, then bring up development and production separately to confirm everything is working as expected

# Part 02: Working with multiple containers

## 1:44:47 Adding a Mongo Container

* Up until now we have been only working with one container hosting the express app
  + Now we will add a mongo database to our application
* In the docker-compose file, add the new mongo database:

|  |
| --- |
| version: "3"  services:    node-app:      build: .      ports:        - "3000:3000"      environment:        - PORT=3000      mongo:      image: mongo      environment:        - MONGO\_INITDB\_ROOT\_USERNAME=alan        - MONGO\_INITDB\_ROOT\_PASSWORD=mypassword |

* + Remember that you can call the service anything you want
  + In the node-app service, we were building our own custom image (with the “.” location), however for the mongo service, we will just use the built-in image, which uses the image property instead
  + You have to pass in some environment variables as part of the mongo image, that being the root username and the root password
* Now do the usual docker-compse up command (do not need --build command):

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose-dev.yml -d |

* Once this finishes, we can look into the mongo container:

|  |
| --- |
| docker exec -it node-docker\_mongo\_1 bash |

* Then connect into mongo with the following:

|  |
| --- |
| mongo -u "alan" -p "mypassword" |

* From here, you can run standard commands such as the following:

|  |  |
| --- | --- |
| **Command** | **Result** |
| db | Shows us what database we are connected to (mongo creates a test database so we have some database to log into) |
| use mydb | Creates a new database and switches to it |
| show dbs | Shows all databases   * mydb is not listed on here because mongodb won’t list databases until there is a document or entry within it |
| db.books.insert({"name": "harry potter"}) | Inserts a document into the books collection |
| db.books.find() | List all documents within the books collection |
| exit | Quit mongo |

* As a shortcut of having to run the "docker exec" command followed by the "mongo" command to login, we can instead add them together and forgo using bash:

|  |
| --- |
| docker exec -it node-docker\_mongo\_1 mongo -u "alan" -p "mypassword" |

* As is, if you shut down the container, you lose the database data, as everything in the previous container is deleted
  + To save this data, we make use of volumes, which help us persist data
* If we wanted to be able to poke around on the database data on our local machine, then we would use a bind mount, but we can just log into the mongo client and run commands to see what I need and don't care about the filesystem, such that an anonymous volume seems better
  + The problem with this is that anonymous volumes are a random bunch of letters and numbers that we don't know what they are for
* What we can do instead is created a **named volume**, which is like an anonymous volume but has a human-readable name
* For the volume, we have to pass in a path within the container, and to get that information, we need to look into the mongo docs
  + From the docs, we can see it is synced with /data/db
* To give an anonymous volume a name, simply supply the name; the format can look as such (this is in the docker-compose.yml file):

|  |
| --- |
| mongo:      image: mongo      environment:        - MONGO\_INITDB\_ROOT\_USERNAME=alan        - MONGO\_INITDB\_ROOT\_PASSWORD=mypassword  **volumes:**  **- mongo-db:/data/db** |

* To summarize:
  + For a bind mount, you provide a path on the local machine to a path on the container
  + For an anonymous volume, you just provide a path to the container that you are interested in
  + For a named volume, you do a name, colon, and then the path to the container
* If you try to run this as is, you will get an error along the lines of no declaration being found in the volumes section
  + It is saying that we have to declare something; we have to declare this volume in another portion of our docker-compose file, because a named volume can be used by multiple services
  + All we have to do is add the volume to the bottom of the file in a new code block; the full docker-compose.yml file now looks like this:

|  |
| --- |
| version: "3"  services:    node-app:      build: .      ports:        - "3000:3000"      environment:        - PORT=3000      mongo:      image: mongo      environment:        - MONGO\_INITDB\_ROOT\_USERNAME=alan        - MONGO\_INITDB\_ROOT\_PASSWORD=mypassword      volumes:        - mongo-db:/data/db  volumes:    mongo-db: |

* From here, now you have to be careful about using the -v flag for the docker-compose down command, because the -v flag deletes both anonymous and named volumes, which we now do not want if we want to persist the database data:

|  |
| --- |
| docker-compose -f docker.compose.yml -f docker.compose.dev.yml down |

* An example of a way to deal with this (as now anonymous volumes will start to build up) is to start the containers you want, and then run the "docker volume prune" command to remove the unused volumes

## 2:01:48 Communicating between containers

* When it comes to interacting with our mongo database, we are going to use a library called "mongoose" which will make it a little easier to talk with our mongo database:

|  |
| --- |
| npm install mongoose |

* + Remember that when you are changing the code, include the --build tag for the docker-compose up command
* Insert the mongoose-based code into index.js; this is the new code in full:
  + If you want to figure out what the IP address of a container is, use the "docker ps" to get all the Docker containers that are running, then run the following to give more detailed information about a container:

|  |
| --- |
| docker inspect <<container name>> |
| docker inspect node-docker\_mongo\_1 |

* + The information we are interested in is around the bottom of the info dump, in the "NetworkSettings" block; in the "Networks" block, you will see a "node-docker\_default" network with an IP address line; grab that
    - In this scenario, the IP address is 172.21.0.2

|  |
| --- |
| const express = require("express");  const mongoose = require("mongoose");  const app = express();  mongoose    .connect("mongodb://alan:mypassword@172.21.0.2:27017/?authSource=admin")    .then(() => console.log("Successfully connected to DB"))    .catch(I => console.logI);  app.get("/", (req, res) => {    res.send("<h2>Hi there..</h2>");  });  const port = process.env.PORT || 3000;  app.listen(port, () => console.log(`Listening on port ${port}`)); |

* + The 27017 is the "default port number"
* If you run the following command, you should see that the database has successfully connected as per the console.log in the index.js code:

|  |
| --- |
| docker logs node-docker\_node-app\_1 |

* We had to use "docker inspect" to go into the container to get the IP address; if we stop and start the container or if we do docker-compose down and up, there is no guranatee we would get the same IP address, and even if it was, we still have to go in and get the IP address ourselves which is somewhat sloppy
* To get around this, Docker has a nice feature that allows us to make it easy to talk between containers, and this feature only exists when it comes to custom networks that get created
* We can look at the Docker networks through the following command:

|  |
| --- |
| docker network ls |

* + We can see a couple of networks, such as the bridge and host networks (which by default come bundled with Docker), as well as node-docker\_default which was created for our application; when we have a custom network (not a default one), we have DNS, so when one Docker container wants to talk to another Docker container, we can use the name of that container or the name of that service to talk to that container
  + Going back to the docker-compose file, we can see that the service for our node app is called node-app, and the service for our mongo app is called mongo
  + We can refer to this container's IP address based off the service name; we can essentially change the code in index.js to, instead of reading the IP address, just provide the service name, which automatically grabs the IP address for us (mongodb://alan:mypassword@**mongo**:27017/?authSource=admin)
* To see how the above works, use the "docker exec -it node-docker\_node-app\_1 bash" command, and run the following commands:

|  |
| --- |
| apt-get update && apt-get install -y iputils-ping |
| ping mongo |

* + The first command is to install the ping command, since Docker containers are minimal and may not come with these standard commands by default
  + The second command shows that it automatically uses DNS to resolve mongo, and it got the IP address
  + Again, **this is only applicable to networks that you create; it does not work with the default (bridge and host) networks**
* If you want to take a look at your networks, run the following:

|  |
| --- |
| docker network inspect node-docker\_node-app\_1 |

* + We can see that the subnet that all the containers are going to use are 172.25.0.0/16 (based on the tutorial video), and we can see in the "Containers" section all of the containers we are using

## 2:12:00 Express Config file

* One thing we don't like is that we are hardcoding the mongoose URL, so create a new folder called "config", and within this create a new file called "config.js"
* From the URL, we need to pull out the username, password, and IP address
  + With the Docker container, we can always use "mongo" so technically we don't need to save this as an environment variable, but in the future, we may want to use some managed service (such as AWS) instead of our Docker container; in that case, we would have to pass in the IP address as an environment variable, so we should just store everything as an environment variable for future planning
* The config.js file will have the following:

|  |
| --- |
| module.exports = {      MONGO\_IP: process.env.MONGO\_IP || "mongo",      MONGO\_PORT: process.env.MONGO\_PORT || 27017,      MONGO\_USER: process.env.MONGO\_USER,      MONGO\_PASSWORD: process.env.MONGO\_PASSWORD  } |

* In our index.js file, we can then import the environment variables and substitute into the mongoose connection string accordingly (changing the string to a template one to accommodate the variables):

|  |
| --- |
| const express = require("express");  const mongoose = require("mongoose");  const {    MONGO\_USER,    MONGO\_PASSWORD,    MONGO\_IP,    MONGO\_PORT,  } = require("./config/config");  const app = express();  const mongoUrl = `mongodb://${MONGO\_USER}:${MONGO\_PASSWORD}@${MONGO\_IP}:${MONGO\_PORT}/?authSource=admin`;  mongoose    .connect(mongoUrl)    .then(() => console.log("Successfully connected to DB"))    .catch(I => console.logI);  app.get("/", (req, res) => {    res.send("<h2>Hi there..</h2>");  });  const port = process.env.PORT || 3000;  app.listen(port, () => console.log(`Listening on port ${port}`)); |

* Now add these environment variables into docker-compose.dev:

|  |
| --- |
| version: "3"  services:    node-app:      build:        context: .        args:          NODE\_ENV: development      volumes:        - ./:/app:ro        - /app/node\_modules      environment:        - NODE\_ENV=development        - MONGO\_USER=alan        - MONGO\_PASSWORD=mypassword      command: npm run dev    mongo:      environment:        - MONGO\_INITDB\_ROOT\_USERNAME=alan        - MONGO\_INITDB\_ROOT\_PASSWORD=mypassword |

* We now have to rebuild the docker container due to passing in new environment variables:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml down |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d |

* Confirm everything is still working with:

|  |
| --- |
| docker logs node-docker\_node-app\_1 |

## 2:21:45 Container bootup order

* When it comes to starting up our Docker containers (especially with docker-compose), we can run into some potential issues; when we spin up both our node container as well as our mongo container, we don't know the exact order that these will get spun up on; Docker will just bring them up at the same time or relatively close to the same time
  + This can lead to issues because if our node container spins up first, it's going to run the mongoose code to try and connect to the database; if the database is not up, it's going to throw an error and crash our application
* We need a way to tell Docker to load up our mongo instance first so we can ensure that it is up and running first before the node container connects to it
  + Docker-compose has a "depends on" field that we can use
* In docker-compose.yml (we want the same behaviour for both production and development environment), under the node-app service, add the following:

|  |
| --- |
| depends\_on:        - mongo |

* + Now the mongo container will always run first; however, this does not fully fix the issue, as the only thing Docker does is that it spins up the container first; it has no idea whether or not mongo has fully intiialized yet or whether the database is actually up and running; this "depends\_on" only helps a little bit
  + Ultimately, there is nothing that Docker can really do in this case; maybe you can have an orchestrator to handle this, but you really want to implement some sort of logic in the application to handle this scenario where the mongo database isn't up and running before the application starts, which usually involves retrying until a successful connection is made
  + Mongoose actually tries for 30 seconds automatically, but will crash out after that, but this is just a note that we need to implement something to handle this
* In index.js, create a function that can assist with this:

|  |
| --- |
| const connectWithRetry = () => {    mongoose      .connect(mongoUrl)      .then(() => console.log("Successfully connected to DB"))      .catch(I => {        console.logI;        setTimeout(connectWithRetry, 5000);      });  }; |

* + This is just an example of how we can implement something to handle this; it will keep checking every five seconds until the mongo database is available and it is succesfully connected to
  + The point of this is that you need to make sure your application handles the logic; do not rely on an orchestrator or on Docker/docker-compose, because none of them can truly guarnatee that the database is up and running; make sure the application is intelligent to handle the scenario
* The full index.js code now looks like this:

|  |
| --- |
| const express = require("express");  const mongoose = require("mongoose");  const {    MONGO\_USER,    MONGO\_PASSWORD,    MONGO\_IP,    MONGO\_PORT,  } = require("./config/config");  const app = express();  const mongoUrl = `mongodb://${MONGO\_USER}:${MONGO\_PASSWORD}@${MONGO\_IP}:${MONGO\_PORT}/?authSource=admin`;  const connectWithRetry = () => {    mongoose      .connect(mongoUrl)      .then(() => console.log("Successfully connected to DB"))      .catch(I => {        console.logI;        setTimeout(connectWithRetry, 5000);      });  };  connectWithRetry();  app.get("/", (req, res) => {    res.send("<h2>Hi there..</h2>");  });  const port = process.env.PORT || 3000;  app.listen(port, () => console.log(`Listening on port ${port}`)); |

* Now tear down and rebuild the container
  + For docker-compose up, if you include the -d flag, it will start all of our containers/services; however, we can tell docker-compose to only start up certain services:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml down |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d **--no-deps** **node-app** |

* + The --no-deps flag is to prevent node-app from also causing the mongo container to run, since that is listed as a dependency of node-app and would normally run alongside it
  + Note that in this situation, the app would timeout as the app will now be continuously tring to connect to mongo which is not running
  + You can run mongo separately too, and with the two services running together the application will work again:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d mongo |

## 2:32:26 Building a CRUD application

* This step will go over creating a blog application
* Create the following folders:
  + models
  + controllers
  + routes
* In the models folder, create a postModels.js file with the following:

|  |
| --- |
| const mongoose = require("mongoose");  const postSchema = new mongoose.Schema({    title: {      type: String,      require: [true, "Post must have title"],    },    body: {      type: String,      required: [true, "Post must have body"],    },  });  const Post = mongoose.model("Post", postSchema);  module.exports = Post; |

* In the controllers folder, create a postController.js file with the following:

|  |
| --- |
| const Post = require("../models/postModel");  exports.getAllPosts = async (req, res, next) => {    try {      const posts = await Post.find();      res.status(200).json({        status: "success",        results: posts.length,        data: {          posts,        },      });    } catch I {      res.status(400).json({        status: "fail",      });    }  };  exports.getOnePost = async (req, res, next) => {    try {      const post = await Post.findById(req.params.id);      res.status(200).json({        status: "success",        data: {          post,        },      });    } catch I {      res.status(400).json({        status: "fail",      });    }  };  exports.createPost = async (req, res, next) => {    try {      const post = await Post.create(req.body);      res.status(200).json({        status: "success",        data: {          post,        },      });    } catch I {      res.status(400).json({        status: "fail",      });    }  };  exports.updatePost = async (req, res, next) => {    try {      const post = await Post.findByIdAndUpdate(req.params.id, req.body, {        new: true,        runValidators: true,      });      res.status(200).json({        status: "success",        data: {          post,        },      });    } catch I {      res.status(400).json({        status: "fail",      });    }  };  exports.deletePost = async (req, res, next) => {    try {      const post = await Post.findByIdAndDelete(req.params.id);      res.status(200).json({        status: "success",      });    } catch I {      res.status(400).json({        status: "fail",      });    }  }; |

* In the routes folder, create a postRoutes.js file with the following:

|  |
| --- |
| const express = require("express");  const postController = require("../controllers/postController");  const router = express.Router();  //localhost:3000/  router    .route("/")    .get(postController.getAllPosts)    .post(postController.createPost);  router    .route("/:id")    .get(postController.getOnePost)    .patch(postController.updatePost)    .delete(postController.deletePost);  module.exports = router; |

* Now go to the index.js file and wire the router up (also npm install body-parser and add that in such that the request body for POST/PATCH/DELETE requests can be properly read):

|  |
| --- |
| const express = require("express");  const bodyParser = require("body-parser")  const mongoose = require("mongoose");  const {    MONGO\_USER,    MONGO\_PASSWORD,    MONGO\_IP,    MONGO\_PORT,  } = require("./config/config");  const postRouter = require("./routes/postRoutes");  const app = express();  const mongoUrl = `mongodb://${MONGO\_USER}:${MONGO\_PASSWORD}@${MONGO\_IP}:${MONGO\_PORT}/?authSource=admin`;  const connectWithRetry = () => {    mongoose      .connect(mongoUrl)      .then(() => console.log("Successfully connected to DB"))      .catch(I => {        console.logI;        setTimeout(connectWithRetry, 5000);      });  };  connectWithRetry();  // This ensures that the body gets attached to the request object  app.use(bodyParser.json());  app.get("/", (req, res) => {    res.send("<h2>Hi there..</h2>");  });  //localhost:3000/api/v1/posts  app.use("/api/v1/posts", postRouter);  const port = process.env.PORT || 3000;  app.listen(port, () => console.log(`Listening on port ${port}`)); |

* To test this using Postman, whenever you are sending a body request, make sure you select the raw format, and change the type to JSON
* Example of parameters to use:

|  |  |
| --- | --- |
| URL | <http://localhost>:3000/api/v1/posts/ |
| Request Type | POST |
| Body | {      "title": "third post",      "body": "body of third post"  } |
| Result | {      "status": "success",      "data": {          "post": {              "title": "third post",              "body": "body of third post",              "\_id": "61ccdb15f3d74b53ec374798",              "\_\_v": 0          }      }  } |

## 2:51:27 Sign up and Login

* Now we can implement a Redis container to handle authentication
* Install bcrypt (npm install bcrypt) for salting/hashing password strings
* In the models folder, create a userModel.js file with the following:

|  |
| --- |
| const mongoose = require("mongoose");  const userSchema = new mongoose.Schema({    username: {      type: String,      required: [true, "User must have a username"],      unique: true,    },    password: {      type: String,      required: [true, "User must have a password"],    },  });  const User = mongoose.model("User", userSchema);  module.exports = User; |

* In the controllers folder, create an authController.js file with the following:

|  |
| --- |
| const User = require("../models/userModel");  const bcrypt = require("bcryptjs");  exports.signUp = async (req, res) => {    const { username, password } = req.body;    try {      const hashPassword = await bcrypt.hash(password, 12);      const newUser = await User.create({ username, password: hashPassword });      res.status(201).json({        status: "success",        data: {          user: newUser,        },      });    } catch I {      res.status(400).json({        status: "fail",      });    }  };  exports.login = async (req, res) => {    const { username, password } = req.body;    try {      const user = await User.findOne({ username });      let isCorrect = false;      if (user) {        isCorrect = await bcrypt.compare(password, user.password);      }      if (user && isCorrect) {        res.status(200).json({          status: "success",        });      } else {        res.status(400).json({          status: "fail",          message: "Incorrect username or password",        });      }    } catch I {      res.status(400).json({        status: "fail",      });    }  }; |

* In the routes folder, create a userRoutes.js file with the following:

|  |
| --- |
| const express = require("express");  const authController = require("../controllers/authController");  const router = express.Router();  router.post("/signup", authController.signUp);  router.post("/login", authController.login);  module.exports = router; |

* Now wire it up in the index.js file:

|  |
| --- |
| const userRouter = require("./routes/userRouters");  . . .  app.use("/api/v1/users", userRouter); |

* You can confirm things are working with parameters such as the following (and then something similar with the login route):

|  |  |
| --- | --- |
| URL | <http://localhost>:3000/api/v1/users/signup |
| Request Type | POST |
| Body | {      "username": "alan1",      "password": "mypassword"  } |
| Result | {      "status": "success",      "data": {          "user": {              "username": "alan1",              "password": "$2a$12$hsWfrj4ZZZorawRRJhMoeexRc63u8MvD2veHFkirMMvItqpbnSm4S",              "\_id": "61cd3e268b96a6178ec7bbf4",              "\_\_v": 0          }      }  } |

## 3:06:57 Authentication with sessions & Redis

**THIS SECTION USES REDIS 3.1.2 AND CONNECT-REDIS 5.1.0, AS OPPOSED TO THE CURRENT VERSIONS OF 4.0.1 AND 6.0.0 RESPECTIVELY, AS THAT IS WHAT THE TUTORIAL USES, AND THE NEW VERSIONS HAVE BREAKING CHANGES THAT MAKE EVERYTHING HERE NOT WORK PROPERLY**

* We will be using express-session to handle authentication
  + Normally we have to ways to handle authentications, that being sessions or JSON Web Tokens (JWTs), but in this situation we will use sessions to showcase Redis
  + With express-session, if you read the docs, you can see that it can work with mongo, Postgres, memory, etc., but again, we will be using Redis
* On Docker Hub, you can see the Redis image we are looking for
  + <https://hub.docker.com/_/redis>
* Add the redis image to the docker-compose.yml file:

|  |
| --- |
| . . .    redis:      image: redis |

* As a shortcut to the docker-compose down and up commands, you can actually just run docker-compose up again (instead of running down first), and docker-compose will be smart enough to detect any changes that we've made and spin up the necessary services
* Looking at the express-session page (<https://www.npmjs.com/package/express-session>), we want the connect-redis package (<https://www.npmjs.com/package/connect-redis>), so follow the npm install for that:

|  |
| --- |
| npm install redis connect-redis express-session |

* When rebuilding containers that are up and running, and you do another "docker-compose up" and you want to rebuild the image, what's going to happen is that the already running image is going to grab the old anonymous volume, which has the old dependencies and packages; we need to force Docker to renew a new anonymous volume (that will have Redis and express sessions)
  + For this, we will need to add the -V option to the docker-compose up command, which will build a new anonymous volume:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d --build **-V** |

* Copy the first part of the API example of the connect-redis package page as a starting point, and build off of that
  + In config/config.js, add the following configurations:

|  |
| --- |
| REDIS\_URL: process.env.REDIS\_URL || "redis",      REDIS\_PORT: process.env.REDIS\_PORT || 6379,    SESSION\_SECRET: process.env.SESSION\_SECRET, |

* + 6379 is default Redis port
  + In index.js, add the Redis logic; this is the full code, roughly:

|  |
| --- |
| const express = require("express");  const mongoose = require("mongoose");  const redis = require("redis");  const session = require("express-session");  const {    MONGO\_USER,    MONGO\_PASSWORD,    MONGO\_IP,    MONGO\_PORT,    REDIS\_URL,    REDIS\_PORT,    SESSION\_SECRET,  } = require("./config/config");  let RedisStore = require("connect-redis")(session);  let redisClient = redis.createClient({    host: REDIS\_URL,    port: REDIS\_PORT,  });  // Establish express routes  const postRouter = require("./routes/postRoutes");  const userRouter = require("./routes/userRoutes");  const app = express();  const mongoUrl = `mongodb://${MONGO\_USER}:${MONGO\_PASSWORD}@${MONGO\_IP}:${MONGO\_PORT}/?authSource=admin`;  const connectWithRetry = () => {    mongoose      .connect(mongoUrl)      .then(() => console.log("Successfully connected to DB"))      .catch(€ => {        console.log€;        setTimeout(connectWithRetry, 5000);      });  };  connectWithRetry();  app.use(    session({      store: new RedisStore({ client: redisClient }),      secret: SESSION\_SECRET,      cookie: {        secure: false,        resave: false,        saveUninitialized: false,        httpOnly: true, // This means that Javascript cannot access it        maxAge: 30000,      },    })  );  // This ensures that the body gets attached to the request object  app.use(express.json());  app.get("/", (req, res) => {    res.send("<h2>Hi there..</h2>");  });  //localhost:3000/api/v1/posts  app.use("/api/v1/posts", postRouter);  app.use("/api/v1/users", userRouter);  const port = process.env.PORT || 3000;  app.listen(port, () => console.log(`Listening on port ${port}`)); |

* + In docker-compose.dev.yml, add the SESSION\_SECRET environment variable:

|  |
| --- |
| - SESSION\_SECRET=secret |

* After building the container with the standard "docker-compose up" command, you will see now then when you send a POST request for logging in that succeeds, you will also get a cookie with a value of 1, with a domain set to localhost and an "Expires" section which expires in 30 seconds, along with some other settings like httpOnly and secure
* If you look into the Redis container and look at its keys (before the 30 seconds from the login expires), you should see the session that was created:

|  |
| --- |
| docker exec -it node-docker\_redis\_1 redis-cli |
| KEYS \* |

* In the authController.js file, in the login method, add the user variable to the session right before the json returns status of 200 when everything is validated:

|  |
| --- |
| req.session.user = user; |

* Do the same with the signUp method:

|  |
| --- |
| req.session.user = newUser; |

* Now let's set up the logic to make sure that for a user to either create or delete or update (or get) a post, they have to be logged in; this can be done by using express middleware, which is a function that runs before the controller
  + This function will have a bit of logic to check that the session object has a user property; if there is, attach to it and forward the request to the controller
  + If there is no user, then return an error saying that you are unauthorized and need to log in
* Create a new folder called middleware, and create an authMiddleware file with the following:

|  |
| --- |
| const protect = (req, res, next) => {    const { user } = req.session;    if (!user) {      return res.status(401).json({        status: "fail",        message: "unauthorized",      });    }    req.user = user;    next();  };  module.exports = protect; |

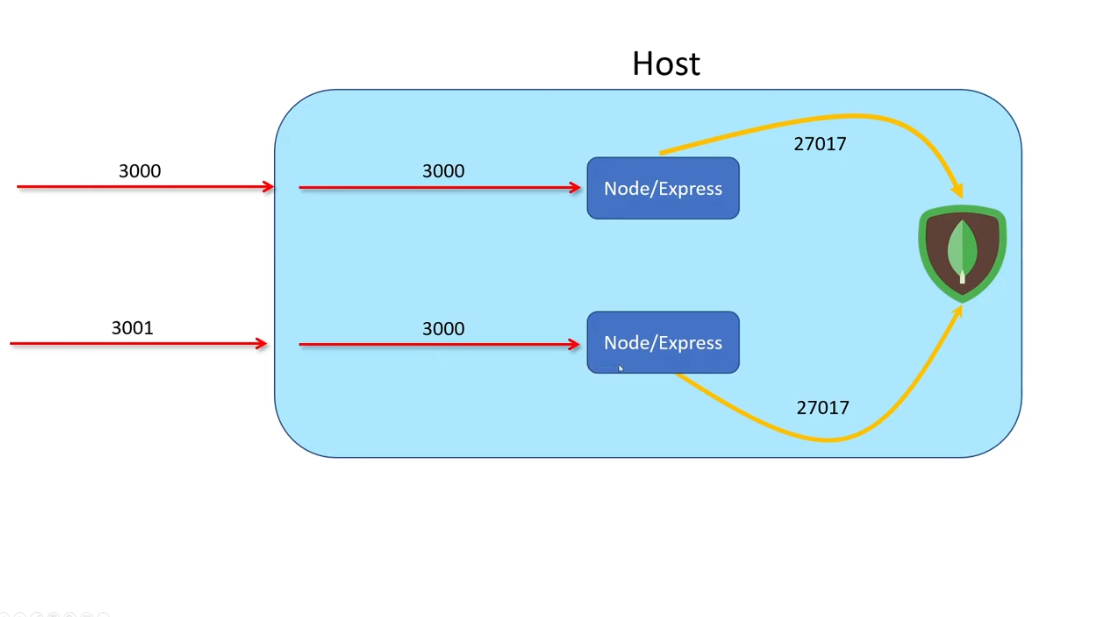
* Now under the post routes, import the protect middleware (full file contents as such):

|  |
| --- |
| const express = require("express");  const postController = require("../controllers/postController");  const protect = require("../middleware/authMiddleware");  const router = express.Router();  //localhost:3000/  router    .route("/")    .get(protect, postController.getAllPosts)    .post(protect, postController.createPost);  router    .route("/:id")    .get(protect, postController.getOnePost)    .patch(protect, postController.updatePost)    .delete(protect, postController.deletePost);  module.exports = router; |

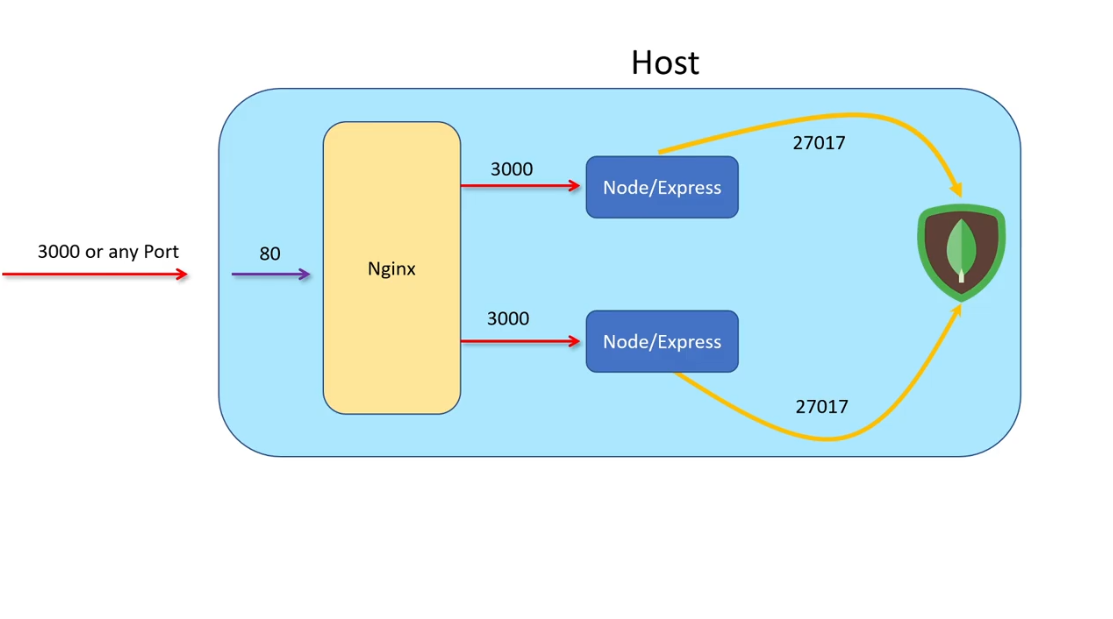
* If the user is logged in (aka. there is a user in the session cookie), then the routes will work, otherwise they will fail with an unauthorized message

## 3:34:36 Architecture Review

* We never opened up a port for our mongo database
  + While we opened up port 3000 for our express server so the outside world can talk to it, we never did that for our mongo datbase, which was on purpose
  + We could technically open port 27017 or whatever port to talk to mongo, but doing this would mean the outside world could talk to it, which is unnecessary as we only need the express application to talk to it, plus this being a security vulnerability
* Docker by default, if you don't open up any ports, already isolates the container
* Now we can address scaling up our containers, increasing the number of node containers to handle increased loads in traffic
  + To do this, you would need to publish a different port; here is an example of a second port being opened:



* It is not feasible to do this manually as the front end should not need to be aware of how many node containers are being run on the backend; the solution to this is to add a load balancer; in this situation, we will be using nginx



* For this situation, we will just use port 80, as that is the default port for HTTP, and is the default port that Nginx listens on
  + For every request that nginx receives, it is going to load balance it to our two node/express instances (or less/more)

## 3:40:48 Nginx for Load balancing to multiple node containers

* Look for the nginx image in Docker hub to be used
* Create an nginx folder with a default.conf file, which will be a basic configuration for our nginx server:

|  |
| --- |
| server {      listen 80;      location /api {          proxy\_set\_header X-Real-IP $remote\_addr;          proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;          proxy\_set\_header Host $http\_host;          proxy\_set\_header X-NginX-Proxy true;          proxy\_pass <http://node>-app:3000;          proxy\_redirect off;      }  } |

* + Listens on port 80
  + Now this is where we actually set it up to redirect traffic to our node/express containers
  + So we say location, and then we provide a url, which is the url of the request that the nginx server receives
  + Put a slash, and then put in all of our configs, with the most improtant one being proxy\_pass
    - For this field, we specify the url of the servers that we want to proxy this traffic to; we want to send this traffic to our express application or our node containers
    - Because our nginix server is also a Docker container, it also has access to DNS, so we can use <http://node-app> because we have the custom network that was created by docker-compose; in docker-compose we can refer to any one of these services by their name, so if we call node-app, it is going to load balance between all of the node-app containers we have
      * Make sure we send it on port 3000 because that is what our express servers are listening on
    - There are a couple of other properties we need to set; because nginx is acting as a proxy, when we actually proxy the original request to our express application, the nginx server is going to strip off a few details; these details may actually be important depending on what your application is doing, and one of the things that nginx does is that you lose the original IP address of the sender
      * We can tell nginx to make sure to forward that to our node application; while our node application isn't making use of that, if you are doing any sort of rate limiting per IP address, these are things that you will need so it is best practice to configure this
      * To ensure that we pass on the original sender's IP, we use proxy\_set\_header X-Real-IP $remote\_addr;
    - Another thing we want to do is pass another setting/flag that will provide us a list containing the IP addresses of every server the client has been proxied through, which is proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;
    - We will add in a few other fields such as proxy\_set\_header Host $http\_host; and proxy\_set\_header X-NginX-Proxy true; and proxy\_redirect off;
    - In this case, all requests are going to be forwarded to our node app; for what we're doing, we are just building a backend; however, if you wanted the nginx server to also handle serving the front end assets, what you would want to do is in your API, for all of your routes, you want to make sure they are listening on api/v1/<<something>>, so that way we know that the nginx server can actually specify that any request that starts with api is meant for our backend, and any request that does not start with api is meant for our frontend
      * If we follow this, then we can add the /api to location; in this case, whatever url is passed for location specifies what the request needs to look like for us to forward it to our node application (any request that comes in starting with /api will be sent to our node app, and anything that doesn't start with /api right now will be dropped)
* Now go to docker-compose.yml file and add the nginx service:

|  |
| --- |
| nginx:      image: nginx:stable-alpine |

* + With this, we no longer have to publish ports for our node application, so we can remove them
* For the development file, add a port 3000:80, and for production, it will be a different port, so do 80:80 for this
* The next thing we need to do is get our configuration file that we built into the nginx container
  + We can either create our own custom nginx image that already has our configuration built in, or we can just configure a volume (bind mount) and just have it sync those two files (this is the route we are going to go)
  + You have to understand a little about where nginx looks for this config, which is in /etc/nginx/conf.d/default.conf, where it expects the file
    - We are going to sync this with ./nginx/default.conf:
    - Also make it read-only as a security check
  + For reference, this is the full docker-compose.dev.yml file:

|  |
| --- |
| version: "3"  services:    nginx:      ports:        - "3000:80"      volumes:        - ./nginx/default.conf:/etc/nginx/conf.d/default.conf:ro    node-app:      build:        context: .        args:          NODE\_ENV: development      volumes:        - ./:/app:ro        - /app/node\_modules      environment:        - NODE\_ENV=development        - MONGO\_USER=alan        - MONGO\_PASSWORD=mypassword        - SESSION\_SECRET=secret      command: npm run dev    mongo:      environment:        - MONGO\_INITDB\_ROOT\_USERNAME=alan        - MONGO\_INITDB\_ROOT\_PASSWORD=mypassword |

* If you search for express proxy, you will see that there is one extra configuration that we need to add into our express application whenever it is sitting behind a proxy; while our application technically does not require it, in a production project you will probably need to add this
  + The configuration in question is app.set trust proxy, which means we are going to trust some of the headers that some of our nginx proxy is going to be adding onto the request
  + Remember we configured the nginx server to add the originating sender's IP address into the header, such that if our express application does need it, we have access to it
  + What we are saying here is telling express to trust whatever nginx is adding onto those headers
  + Add this to index.js:

|  |
| --- |
| app.enable("trust proxy"); |

* Now we can scale up our application (add a second node instance); first tear everything down, then bring it back up but this time using the --scale flag:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d -**-scale node-app=2** |

* The next thing to do is in one of the routes, you can do something like adding a console.log to one of the routes to verify that nginx is actually load balancing the request to all of our node instances
  + You can do this by split screening two terminals, with one terminal being for instance 1 and the other being for instance 2
  + You can check it with separate commands as such:

|  |
| --- |
| docker logs node-docker\_node-app\_1 -f |
| docker logs node-docker\_node-app\_2 -f |

* + The -f flag stands for follow
* You should see that if everything was done correctly, only one of the containers will print out the console.log upon sending the request, and then the next time, the other container will send it which is successfully load balancing

## 3:54:33 Express CORS

* One last thing to do is to enable CORS, which allows your front-end to run on one domain and your back-end to run on another domain; by default, if they are on two different domains and the front-end sends a request to the back-end, the API will reject that request from our front-end
* To allow these to be running on different domains, we have to configure CORs so that different domains can access our API
* Install CORS:

|  |
| --- |
| npm install cors |

* + Because we added a new package to our package.json file, we will have to rebuild our image
  + Remember that by default, when we run a docker-compose up when we are already up and running, if we have an anonymous volume like we do in docker-compose.yml (i.e., /app/node\_modules), it's going to use our old anonymous volume that only has the node\_modules before we ran the up command; to get the new cors package added in, we need to pass in the -V flag which recreates a new anonymous volume

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d --build -V |

* Now go to index.js and import the cors library to be used

|  |
| --- |
| const cors = require("cors");  . . .  app.use(cors()); |

* + We have config options that we can pass into cors to tweak the configuration, but we can just leave the default settings

# Part 03: Moving to Prod

## 3:57:44 Installing docker on Ubuntu(Digital Ocean)

* The example uses Digital Ocean, although you can use any other serivce like AWS/EC2/Azure if that is your preference
* In the context of Digital Ocean, this is initialized as a droplet under Ubuntu; once it is completed (which does incur a monthly cost), an IP address is provided, which you can access in a terminal through ssh root@<<IP address>>, continue connecting, and use the password you supplied
* After being logged in, the first thing to do is to get Docker installed (look at the Install Docker on Ubuntu documentation)
  + If you look at get.docker.com, there is a script that installs Docker for you automatically:

|  |
| --- |
| # This script is meant for quick & easy install via:  # $ curl -fsSL <https://get>.docker.com -o get-docker.sh  # $ sh get-docker.sh |

* + This will download a file called get-docker.sh, and then run it
* Verify that docker is installed:

|  |
| --- |
| docker --version |

* Docker-compose is not installed by default, so look for installation instructions for that for Linux (<https://docs.docker.com/compose/install/>):

|  |
| --- |
| sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose |
| sudo chmod +x /usr/local/bin/docker-compose |

* + Now running docker-compose -v should provide version information

## 4:03:21 Setup Git

* In Github, click the + icon to create a new repository and give it whatever name you want (node-docker), and make it private or public (in this case it will be public)
* Follow the instructions to get the repository into Github
* Create a .gitignore file, and make sure the node\_modules folder is added here
* Go through the following commands to get this added:

|  |
| --- |
| git init |
| git add --all |
| git commit -m "first commit" |
| git branch -M main |
| git remote add origin <https://github>.com/AltiV/node-docker.git |
| git push -u origin main |

## 4:05:37 Environment Variables on Ubuntu

* Go to docker-compose.dev, and you will see you see you need some environment variables such as MONGO\_USER, MONGO\_PASSWORD, and SESSION\_SECRET, MONGO\_INITB\_ROOT\_USERNAME, and MONGO\_INITB\_ROOT\_PASSWORD
  + You definitely do not want to accidentally push any of your production secrets or configs or passwords into Github, as if you put the environment variables into the docker-compose files, they are all pushed into Github
* What we will do is we are going to get all the environment variables from the machine that Docker is running on
  + From the Ubuntu machine, we will configure the environment variables, and from there, Docker will know what those values of and pull them from the host Ubuntu machine
* From a configuration perspective, copy those environment variables and paste them into docker-compose.prod.yaml, and change as following (the file contents are below):

|  |
| --- |
| version: "3"  services:    nginx:      ports:        - "80:80"    node-app:      build:        context: .        args:          NODE\_ENV: production      environment:        - NODE\_ENV=development        - MONGO\_USER=${MONGO\_USER}        - MONGO\_PASSWORD=${MONGO\_PASSWORD}        - SESSION\_SECRET=${SESSION\_SECRET}      command: node index.js    mongo:      environment:        - MONGO\_INITDB\_ROOT\_USERNAME=${MONGO\_INITDB\_ROOT\_USERNAME}        - MONGO\_INITDB\_ROOT\_PASSWORD=${MONGO\_INITDB\_ROOT\_PASSWORD} |

* To set an environment variable on a Linux machine, type in the following:

|  |
| --- |
| export <<variable>>=<<value>> |
| export SESSION\_SECRET="hello" |

* Using "printenv" will print out all the environment variables
* We could go adding all the variables one by one, but this is a slow process, and they won't actually persist through machine reboots, so a way to deal with these is to create an environment file
  + Technically you should not be doing anything under the root user (based on security based practices), but for now we will just do this for simplicity
  + Pick a location, recommended nowhere near where the application code is going to be stored so you don't accidentally push this into git
  + In the Linux machine (again, this is where you are linked to what was created back in the Digital Ocean Droplet or similar):

|  |
| --- |
| vi .env |

* In this file, store all the environment variables:

|  |
| --- |
| NODE\_ENV=development  MONGO\_USER=alan  MONGO\_PASSWORD=mypassword  SESSION\_SECRET=secret  MONGO\_INITDB\_ROOT\_USERNAME=alan  MONGO\_INITDB\_ROOT\_PASSWORD=mypassword |

* Then if you do a "ls -la", you will see a ".profile" file; if you open this up with "vi .profile" and go to the bottom, add a line to export the environment variables:

|  |
| --- |
| set -o allexport; source /root/.env; set +o allexport; |

* + This loops through all the environemnt variables that we set, and set them on the machine
  + These changes won't take effect until we close out our terminal session and reopen it

## 4:14:12 Deploying app to production server

* Since we have made some changes, add them to git:

|  |
| --- |
| git add --all |
| git commit -m "env changes" |
| git push |

* Go to the production server and create a folder called app, and clone the git repo:

|  |
| --- |
| mkdir app |
| cd app |
| git clone <https://github.com/AltiV/node-docker> . |

* + The "." is to clone into current directory
* Now in the production machine, run a docker-compose command:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d |

* If any mistakes were made prior to this, then you will have to do the whole git add/commit/push commands again, but you can use "git pull" instead of "git clone" for the last part
* To confirm everything is working after the docker-compose finishes running, you can do a standard "docker ps" to check for the running containers
* To test things are working, pull up the IP address of the Digital Ocean droplet (or whatever you are using) (example from video):



* Go to Postman and put in the URLs as standard:

|  |
| --- |
| <http://104>.236.81176/api/v1 |

* Remember that since we deployed this on a different server, there is nothing in the database, so we will have to signup first

## 4:18:57 Pushing changes the hard way

* Whenever a change is made, go through the standard git add/commit/push commands, then in the production machine, go the the app folder and do git pull
* Because this is the production environment, it is not going to automatically sync the code changes; we have to rebuild the image and create brand new containers
  + We can do a docker-compose down and docker-compose up, or just do a docker-compose up
    - If you do docker-compose down and docker-compose up, there will be a bit of outage time
  + Using the docker-compose up command, the node-app is not actually updated; that is because docker-compose is "dumb" and only checks to see if there is an image under the expected name, not knowing if the image is out of date
  + Therefore, make sure to include the --build flag (as we have been doing usually, but this is just a reminder):

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --build |

* Even though this whole process works, there are some things that are not so great:
  + First of all, when you do the up -d --build, it will check all of the containers and all of the services to see if anything has been changed; we know that the only thing is going to be changing when we change our source code is going to be the node-app container (and not the mongo/redis/nginx contianers)
  + To handle this, just add the service-name specificailly to the command (node-app), but it still checks mongo, as within docker-compose, node-app depends on mongo
    - Within docker-compose, any time you specify a service to be rebuilt, it will have no idea if any of the dependencies change, so it has to rebuild them regardless
  + There is a way around that by passing in another flag, that being the --no-deps flag:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --build --no-deps node-app |

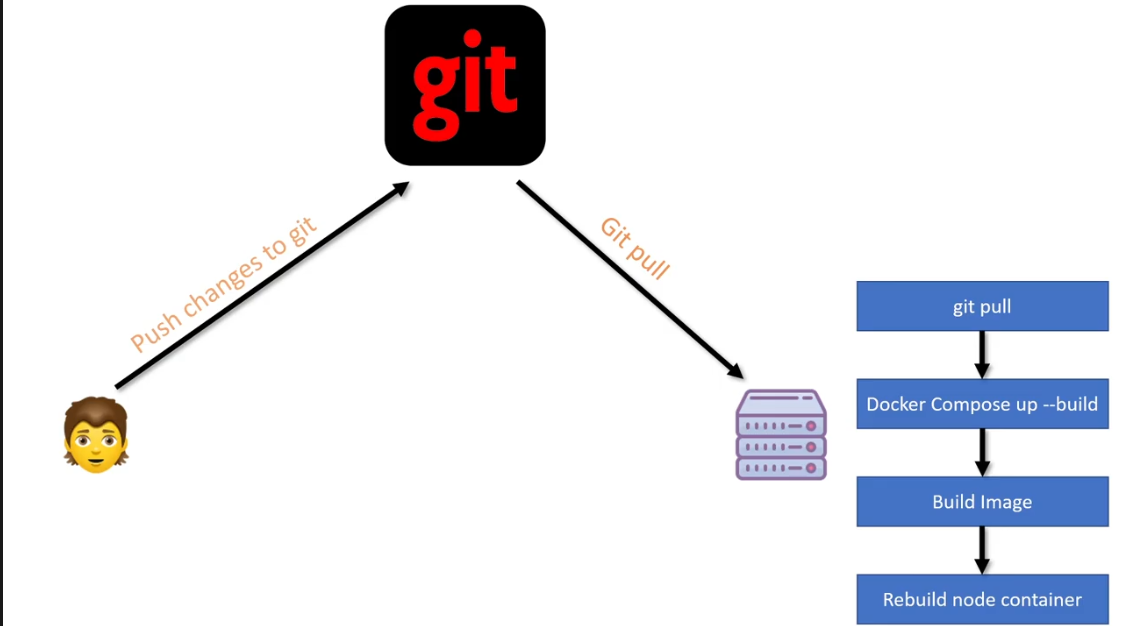
## 4:25:58 Rebuilding Containers

* There may be an instance where you want to rebuild a container regardless of if any changes have been made
* If we just run the standard docker-compose up command, nothing will change because the containers are already considered as up-to-date
* You can get around this by adding the --force-recreate flag:

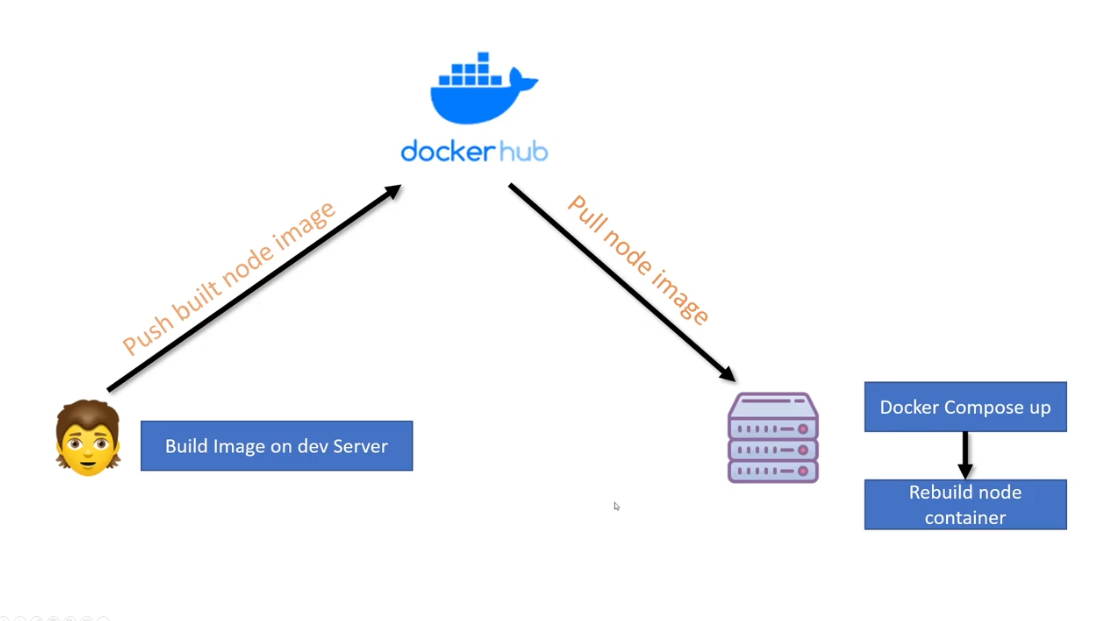
|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --build --no-deps --force-recreate node-app |

## 4:27:32 Dev to Prod workflow review

* The main issue of what we have been doing so far is that you should be never be building an image on the production server, as building an image takes resources (CPU cycles/memory)
  + For our application, it is small so it doesn't take that much, but as it grows, it will require larger and longer build times
  + In a production server, you could end up starving your production traffic because all of the compute power and memory is going to building the image
  + The production server should only be meant for one thing, and that is to handle production traffic, and nothing else
* Therefore, we want to move away from this workflow
* Before:



After:



* By building the image on the dev server, we no longer have to worry about building it on the production server

## 4:30:50 Improved Dockerhub workflow

* To implement a new workflow, create an account on Docker Hub
* After this, create a new repository (name of node-app as default)
  + You get unlimited public repositories and one private repository
* Now let's push the image we have on the development server up to the repository so it can store our final image
  + Have a look at the images with "docker image ls" and grab the latest repository
  + When you push an image to Docker Hub, it needs to have a very specific name; copy the entire name (<<username>>/<<image name>>)
  + To rename an image on your machine, use the following:

|  |
| --- |
| docker image tag node-docker\_node-app <<username>>/<<image name>> |
| docker image tag node-docker\_node-app sloppynetworks/node-app |

* + sloppynetworks/node-app is just the example used in the video, but substitute your own container name if you are following this on your end
  + What this actually does is it copies the image and gives the copied image the name you provided
* Now you can push the image by logging in and using the expected name:

|  |
| --- |
| docker login |
| docker push sloppynetworks/node-app |

* If you go back to Docker Hub and refresh the page, you should see that the image was successfully pushed
* Before our production server can actually pull this image, we have to tell docker-compose that we want to use this image moving forward
  + We still need to build the image ourselves with docker-compose, but we also need to be able to tell it that when we run the application, we want to use this specific image from the repository
* In docker-compose.yml, under node-app, pass the image property:

|  |
| --- |
| node-app:      build: .  **image: sloppynetworks/node-app**      environment:        - PORT=3000      depends\_on:        - mongo |

* Push the changes to git, pull in the production server, and now run docker-compose without most of the special flags (i.e., without building):

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d |

* It rebuilt the image, as now the image is now from Docker hub (ex. sloppynetworks/node-app), and nothing's changed
* Let's say we make some more changes in the development environment; to push them, remember that we want to build an image on our local machine, and then push it to our repository on Docker Hub, and then pull this into the production server
* The first step on the development machine, instead of up/down, will now be build

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml **build** |

* + Remember to use the production file, as we are building for production and not devleopment
  + This will look through all of our services that allow us to build an image
    - Right now we only have one custom image, that being node-app
    - If we had more than one, then it would build all of the images
* Now that we have this image, we can then push it up to Docker Hub
  + By default, this will build all of the associated images; if we only wanted to build the image for one of the services, then pass in the specific name of the service
* Now push to Docker Hub; you have the option of pushing all the images for all of the services, or specify just the services we want to push:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml push |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml push node-app |

* Now go to the production server, and instead of running the docker-compose up command, pull the image, and start the container with the new image:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml pull |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d |

Again, to specifically only pull one specific image because we may not want to update the other ones, just include --no-deps and the name of the container:

|  |
| --- |
| docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --no-deps node-app |

## 4:46:10 Automating with watchtower

* When we push an image to Docker Hub, we can have the production server automatically detect a new image was pushed, and pull that new image by using something called Watchtower
  + Some people may like this being automated but some people may want to handle this themselves, so this is just a tool to be aware of
* In the Quick Start section of the Docker Watchtower page, which is a special container that periodically watches Docker Hub for a specific image, and if it sees a new image be pushed out, it will pull that image automatically and restart the container
  + Essentially it is a container that handles the automation of the other containers
* In the production server, run the following:

|  |
| --- |
| docker run -d --name -watchtower -e WATCHTOWER\_TRACE=true -e WATCHTOWER\_DEBUG=true -e WATCHTOWER\_POLL\_INTERVAL=50 -v /var/run/docker.sock:/var/run/docker.sock containrrr/watchtower app\_node-app\_1 |

* + Link to Watchtower info: <https://containrrr.dev/watchtower/>
  + Trace: Verbose logging
  + Poll interval: How frequently we should watch Docker Hub
  + At the end, make sure to specify the services/containers that you want to watch out for
* If you do a "docker ps", you should see the new watchtower container running
* If you do a "docker logs watchtower -f", you should see some notes about it retrieving running containers, and that the first check will be performed in X seconds
* If you go back to the development server, make changes, and do the docker-compose build/push commands, and go back to the production server and look at the watchtower logs, you will see the changes of it checking a bunch of stuff and pulling the image
  + The reason watchtower checks for credentials is that there is a possibly that one of the containers uses an image from a private repository; we would have to make sure to use a docker login in the production server prior if that were the case

## 4:56:06 Why we need an orchestrator

* If you were using watchtower, just delete it as we will not be using it going forward
* With regards to our current workflow, at the end of the day, we need to recreate the container (that is, tear down the current container, build a brand-new container with a brand-new image, and then start that container)
* During that window of tearing down and building up, we are going to experience a network outage/our application being down until the new container is built, which will result in a loss in production traffic
* Docker-compose can somewhat achieve rolling updates/doing upgrade process/pushing new changes to the production server without experiencing any loss, but they are all somewhat hacky methods that should not really be used in a production environment
  + This is because docker-compose isn't meant for that; **it is not a container orchestrator**
  + It is nothing more than a file that maps out to different docker run commands, as a service is nothing more than a container that is created with docker run
* An option to achieve lossless upgrades and rolling updates is using one of the popular container orchestrators, such as **Kubernetes**
  + However, as Kubernetes would require its whole separate tutorial series and additional days' worth of information, we will instead use a built-in container orchestrator that comes with Docker called Docker Swarm
  + The main reason we are using Docker Swarm instead of Kubernetes (beyond the time sink) is to just show the purpose of a container orchestrator
* Some differences between Docker-compose and Docker Swarm is that docker-compose is ultimately not a container orchestrator; it cannot handle some of the more important lifecycle events when it comes to spinning up, deleting containers, and being able to handle rolling updates, which Docker Swarm can
* We can also only use docker-compose to deploy containers onto one server; if we wanted to distribute the express containers (such as five or six of them) across multiple servers so that if one goes down, we'll have some redundancy with other servers being able to pick up the slack, we cannot do that with docker-compose
* Docker-compose is just a bunch of docker run commands contained in a .yml file
* Docker Swarm has logic and gives us the ability to not only spin up containers, but distribute them to as many servers as we want
  + So if we have 5, 10 production servers, we can spread them out across all of our servers, we can handle the update process so if we need to push a new image to our production server, Docker Swarm can then spin up new contianers, update those containers, and then only once we verify those containers are up and running, we can then delete the old containers
* Docker Swarm gives us a multi-node environment, which means we can use multiple servers to deploy our applications; we don't need to run everything on one server
* Each server within a Docker Swarm is referred to as a node, which is separated into manager nodes and worker nodes
  + A manager node handles all the brains behind everything, and pushes out tasks to the worker nodes
  + The worker nodes carry out those tasks that it receives
  + A manager node can be both a manager node and a worker node
* Docker-compose is ultimately a development tool, not a production tool

## 5:03:32 Docker Swarm

* Docker Swarm is shipped with Docker, but is disabled by default
* To enable Docker Swarm, just run the following in the production server
  + You will need to grab the IP address; for Digital Ocean you will have two IP addresses (one on eth0 and eth1); just grab the one with the public facing IP:

|  |
| --- |
| docker swarm init --advertise-addr <<IP address>> |

* You will get a message along the following: "Swarm initialized: current node (<<id>>) is now a manager."
* If you wanted to add more nodes into the swarm, there are two commands you can use (the first one adds a node as a worker, and the second one adds a node as a manager); we will stick with just one node for now to keep things simple:

|  |
| --- |
| docker swarm join --token <<token>> <<ip-address>> |
| docker swarm join-token manager |

* Docker Swarm is very similar to doing regular Docker conainers such as run and create
  + Instead of working with containers, you work with services, which are pretty much like containers
* Run the following to get access to all swarm commands:

|  |
| --- |
| docker service --help |

* We can do everything by running docker service commands, but this is tedious, similar to docker run commands and having to remember all the flags
* When we wanted to automated Docker run commands, we just put all of them into a compose file; with Docker Swarm, you can do the same thing, and even use the compose files that we already have and just add an additional section that is swarm related
* If you go to the reference section under docker compose and search for the deploy section, it will give all the information we need (<https://docs.docker.com/compose/compose-file/compose-file-v3/>)
* As an example, the replicas define how many instances of a specific service you want to run
  + If you set replicas to 6 for the node-app, it will give us six containers
* The restart policy determines how and when we restart the container, and if so, how long we should wait
* In UPDATE\_CONFIG, this is what we are most interested in as we are trying to create a way to update our application without experiencing any loss
  + There is a flag called "parallelism", which sets the number of containers to update at a time (ex. if you set it to 2, it will update two containers at a time)
  + Delay is the time to wait between updating a group of containers
  + Failure\_action determins what to do if an update fails
* Under the docker-compose.prod.yml file (we do not need this in development), add a deploy section under node-app with the following:

|  |
| --- |
| deploy:        replicas: 8        restart\_policy:          condition: any        update\_config:          parallelism: 2          delay: 15s |

* Do the standard git add/commit/push on development server and git pull on production server, which will now have the updated docker-compose file
* Tear down everything with docker-compose down, and now to deploy the application using Docker Swarm, use the following:

|  |
| --- |
| docker stack deploy -c docker-compose.yml -c docker.compose.prod.yml myapp |

* + A stack name is just the name of the application (all of your services bundled together), which is called "myapp" here
* You will see that it will create all of our services, which includes a default network and the four services
* Run the following to list out all of the nodes within Docker Swarm:

|  |
| --- |
| docker node ls |

* Run the following to list all of the stacks:

|  |
| --- |
| docker stack ls |

* Run the following to list all the services in a stack:

|  |
| --- |
| docker stack services myapp |

* + Note that of the four noted services (this is equivalent of docker ps), and that for node-app, you should see eight individual containers (8/8); you can verify that with docker ps to see most of them are node-app containers
* Run the following to list all of the services across all stacks (keep in mind we only have one stack, so we see the same exact output as we saw before):

|  |
| --- |
| docker service ls |

* When it comes to creating/updating/deleting a service, Docker Swarm generates a task and pushes it to a worker node so that the worker can perform the task; you can list out all of the tasks for the stack with the following:

|  |
| --- |
| docker stack ps myapp |

* + You can see a task got generated for each of the methods to provision the container

## 5:16:13 Pushing changes to Swarm stack

* Now make a change to the application and see if we can update the production server with those changes but with a rolling update methodology using Docker Swarm so we experience minimal to no loss
* After changes, do the usual build and push to Docker Hub
* Now go to the production server and run the "docker stack deploy" command, which will show it is updating the services
* Now check the "docker stack ps myapp" command, and you should see that two of the containers have been shutdown, which is due to the parallelism
  + If you wait a bit and run the command again, now you will see two more containers shutdown and two new containers created
* While it is updating, if you keep sending requests, you may hit some containers that are updated and some containers that aren't updated
  + Even with an orchestrator, you can't guarantee no downtime, as you may sometimes hit an error with the request spam
* For each of the containers, you will eventually see a bunch of tasks; for each of the containers, one with a shutdown command and one with a running command