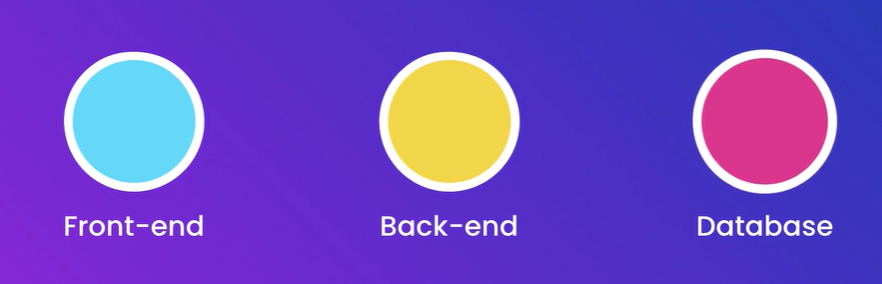


**Introduction**:

In this section we will learn about running multi container applications. So we will have a real world application with three building blocks.

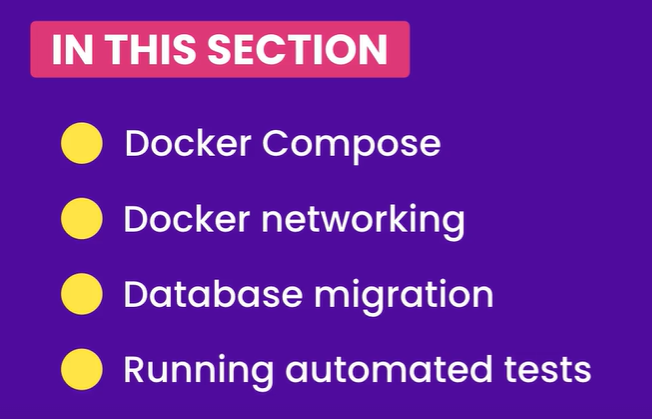


Front – End 🡪 React

Back – End 🡪 Node

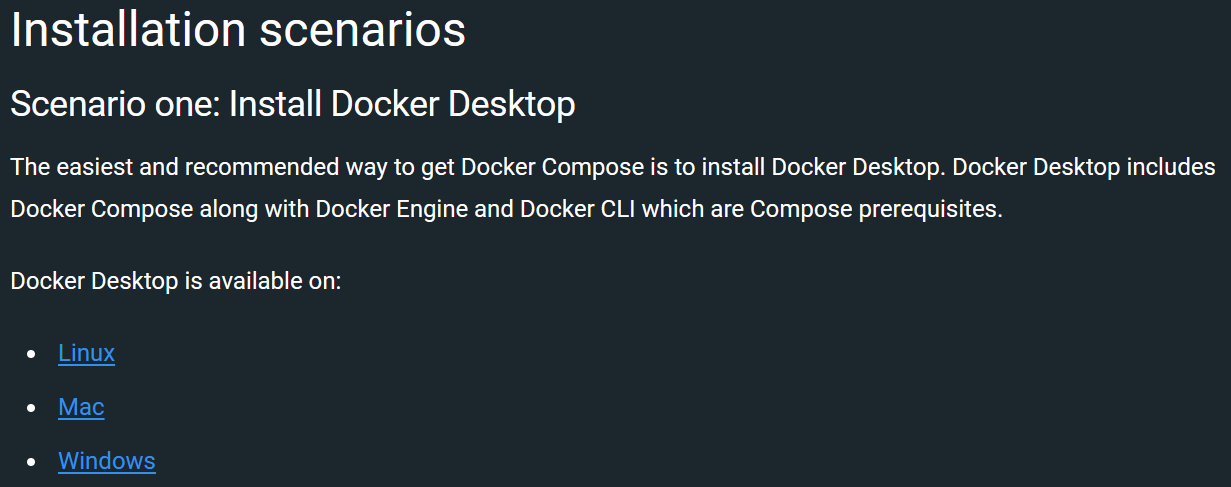
Database 🡪 MongoDB

We will learn about,



**Installing Docker compose**:

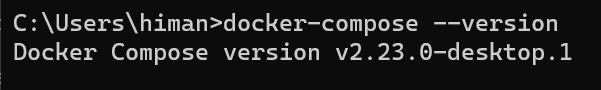
In this section, we are going to use a tool called ***Docker Compose***, which is built on top of Docker engine. It makes I incredibly easy to start applications with multiple containers.



So if we have docker desktop already installed we don’t have to worry about installing docker compose.

We can verify it by running this command,

docker-compose --version

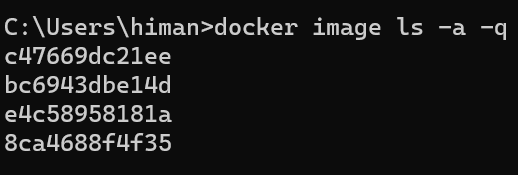


**Cleaning up our workspace**:

There are already some images on our machine and stopped/ running containers, so we need to clean them all to have a fresh start.

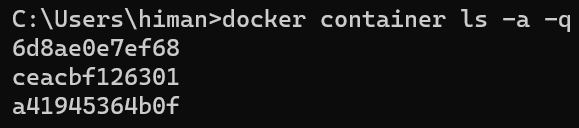
There is a cool trick *to see all the image ID’s*:

docker image ls -a **-q** (*q argument gives the id*)



and *to see all the container ID’s*:

docker container ls -a **-q**



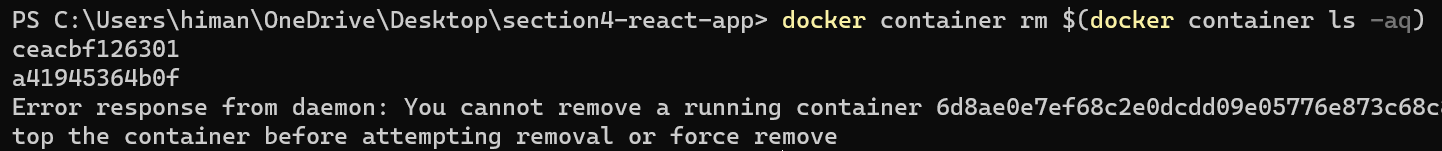
This information can be used as a argument to delete all the images and containers when paired with *rm* command like this,

**BIG SIDE NOTE**:

Next command only work with ***PowerShell*** *or* ***BASH***:

docker container rm $(docker container ls -aq)

Commands involving $ can’t be used inside windows default terminal.



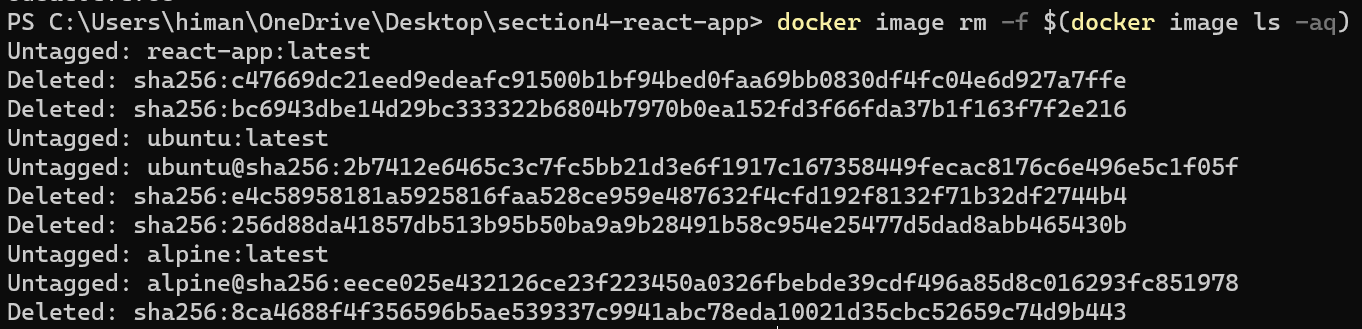
Some containers are still running so need to remove them using – f option,

docker container rm -f $(docker container ls -aq)

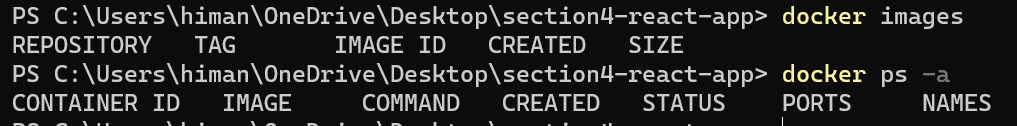
All containers are removed.

Now let’s remove all the images,

docker image rm -f $(docker image ls -aq)

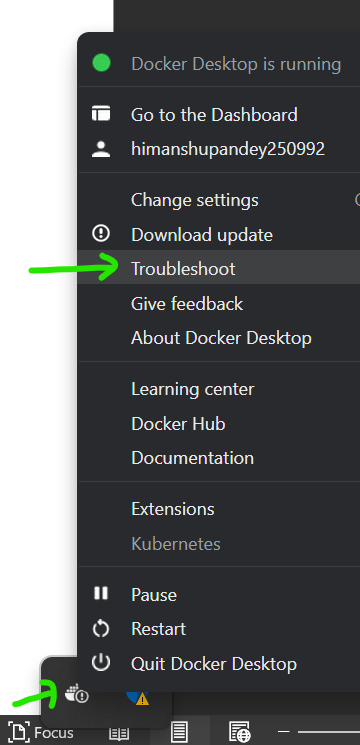


As a result we have a clean workspace now, no images and no containers.

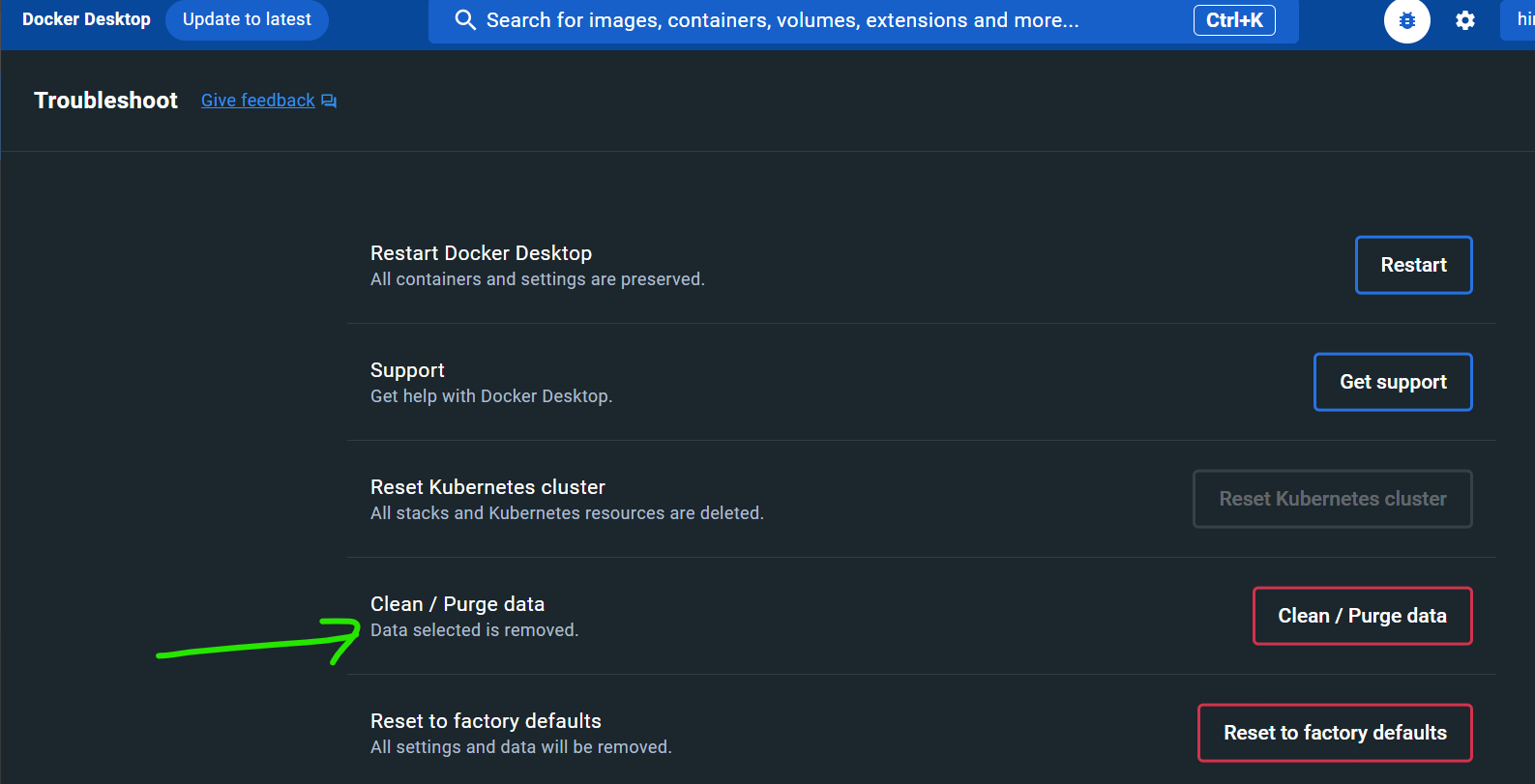


There is another way to achieve this,

Open up Docker menu from the *docker icon* and click on *troubleshoot*.

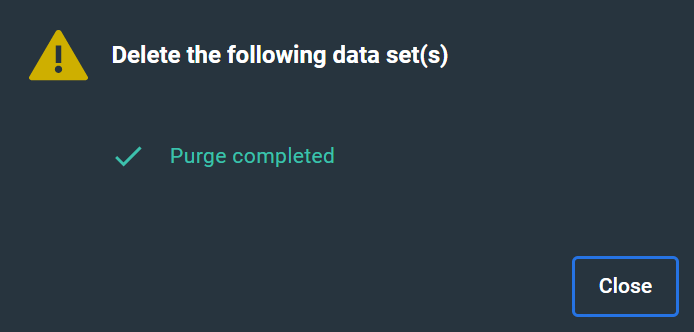


On this page we have some useful utilities including *Clean / Purge data*.



This will essentially remove everything from docker like images, containers, volumes and so on.

After purging,

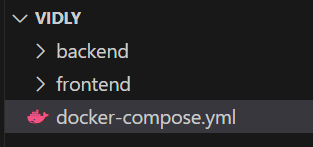


Need to wait for some time for docker engine to restart.

**The sample web application**:

In this section we will look at real world application with multiple building blocks (*frontend, backend and a database*).

Download the sample application.



Inside backend folder we have our node project which starts a webserver at port 3001.

Then we have a frontend project which is a react application that talks to the backend.

If we need to run this application outside the docker there are a number of steps that we need to follow.

🡪 We check out our code from GitHub repository.

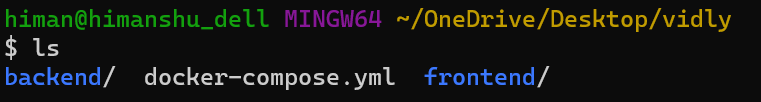
🡪 Go to backend folder, install all dependencies and start the webserver.

🡪 Open up another terminal window and repeat the same steps with frontend project.

🡪 Download and Install MongoDB on our machine.

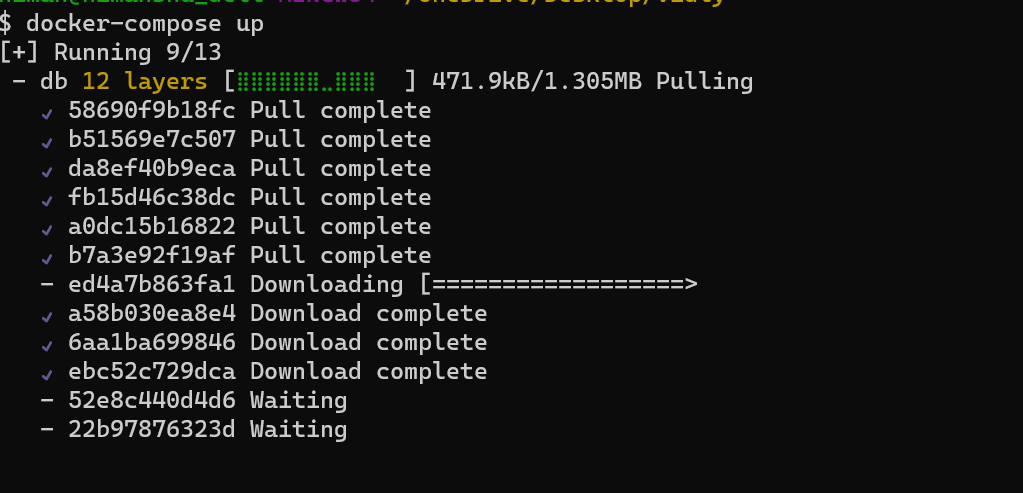
So there are a lot of steps we need to follow as soon as we check out our code from GitHub repository.

With Docker we don’t have to do any of these steps. All we have to do is run a single command.



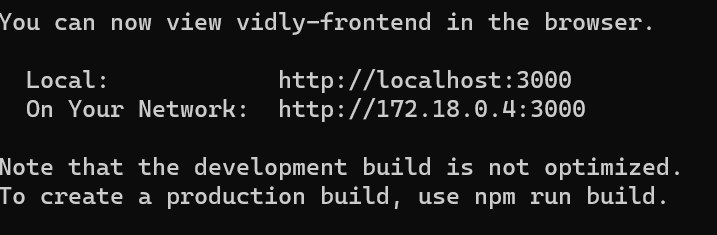
Notice a *docker-compose.yml* file inside our sample application folder. This file is *used for composing a multi container application*.

Once we have this file in our project directory, we can simply run docker-compose up command.

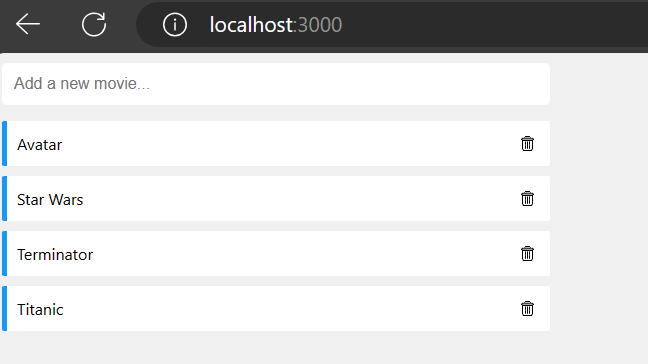


Now Docker is automatically downloading a particular version of MongoDB, all the layers and at the same time downloading and installing all the dependencies of our frontend and backend project, start webservers and run automated tests, ALL IN THIS WINDOW.

Now our application is up and running,



We can access it at port 3000.



We have a mini application for managing a list of movies.

The beauty is that our database is populated with these movies as part of bringing up our application. We did not had to manually insert these movies into the database.

So *we have a migration script populating our database and docker automatically executed our migration script as part of bringing up this application*. This is a very common real world scenario.

Here we can add new movies by typing and entering into the Text box and we can also delete these movies.

So we brought up this application by using a single command.

**JSON and YAML formats**:

In the root of this project let’s create a *data.json* file.

A JSON is a human readable language for representing data. So in this JSON file we have an object or an array.

Let’s say we want to represent a course. A course can have multiple properties in the form of key – value pairs.

{

    "name":"The Ultimate Docker course",

    "price": 149,

    "is\_published": true,

    "tags": ["software", "devops"],

    "author":{

        "first\_name": "Himanshu",

        "last\_name": "Pandey"

    }

}

Keys are always surrounded in double quotes.

So we have a JSON file now let’s see how can we convert this into YML file.

YAML or YML is another language for representing data but it has less clutter then JSON. It’s easier to read.

---

name:The Ultimate Docker course

price: 149

is\_published: true

tags:

  - software

  - devops

author:

  first\_name: Himanshu,

  last\_name: Pandey

Notice we did below changes here:

🡪 On the top we have three hyphens to indicate the beginning of YML file.

🡪 In YML we don’t use curly braces to indicate hierarchy instead we use indentation (*like in python*).

🡪 Replace all the double quotes with nothing.

🡪 Remove commas to separate different key – value pairs.

🡪 To represent a list or an array, we need to use hyphens.

🡪 To represent an object, we can remove { } and replace it with indentation.

So compared to JSON, YML is easier to understand.

Note: We don’t use YML all the time because parsing YML files is a little bit slower compared to JSON files. The reason is that the parser does not know if it’s a string or a number (*in json “ ” double quotes represent string*), so it has to read everything as a string and then reevaluate it.

*Quite often we use YML files as configuration files and JSON for exchanging data between multiple computers like a client and a server*.

**Creating a Compose File**:

Let’s see how can we create a compose file from scratch. So we will rename the existing file by adding a \_ at the beginning of file name (*so that we can set it aside and create a new compose file from scratch*).

Create a new file called docker-compose.yml (*make sure to use all lowercase*) otherwise ***docker-compose*** utility is not going to find this file. This is the default name docker-compose assumes.

First thing we need to set is the *version* property. What version should we use.

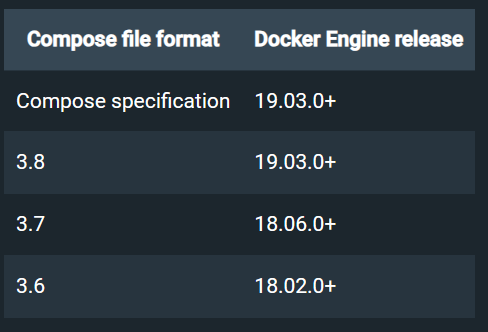
Search for docker compose file from below link…

[Compose file versions and upgrading | Docker Docs](https://docs.docker.com/compose/compose-file/compose-versioning/)

The version we should use in docker-compose file depends on the version of docker-engine and docker compose CLI that we have installed.



Here on this page we can see various compose file formats and their compatibility with docker engine.



Since we are using latest version of docker-engine so we will use latest compose file format to have access to latest features.

version: "3.8"

Here we need to wrap this version inside double quotes otherwise it will be evaluated as a number but *docker-compose expects this version value as a string*.

Now in this file we need *to define various building blocks or services of our application*. So we have a property called ***services***.

So what services do we need here. Well our application has a frontend, a backend and a database (*each application can have different moving parts and we can define them here*).

version: "3.8"

services:

  frontend:

  backend:

  database:

These names are arbitrary and we can call them anything like changing database to db, backend to api and frontend to web.

version: "3.8"

services:

  web:

  api:

  db:

Well **“***the idea here is that we are defining various services and telling docker how to build images for each service and how to run these images***”**

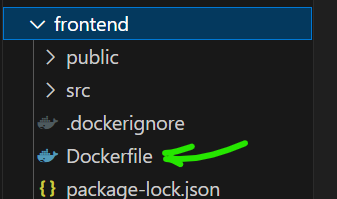
So *here we are going to have properties and the value of these properties will eventually be used when running our containers*.

In the previous section we had to manually run our containers using *docker run* command and here we used parameters like -p (*for port mapping*) or -v (*for volume mapping*). We also had to specify an image name(*like react-app*).

All these values can be defined in our compose file, so we don’t have to manually start our containers. Docker – compose will take care of starting our containers under the hood.

So for each service we need to tell docker, how to build an image for that service.

We can use the ***build*** property and tell docker – compose where it can find a Dockerfile.



In the backend and frontend directory we have a *Dockerfile*.

FROM node:14.16.0-alpine3.13

RUN addgroup app && adduser -S -G app app

USER app

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]

This Dockerfile is almost identical to the one we created in the previous section.

Here we have (*from the top*),

🡪 A node image.

🡪 Created a user.

🡪 Setup our working directory.

🡪 Copied all the files and install dependencies.

🡪 Expose port 3000

🡪 Start the webserver.

We have a similar Dockerfile in our frontend project,

FROM node:14.16.0-alpine3.13

RUN addgroup app && adduser -S -G app app

USER app

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3001 🡪 Except port number

CMD ["npm", "start"]

So each service should have its own Dockerfile.

Back to the compose file. We will set the build property for each service as the location of their respective Dockerfile.

version: "3.8"

services:

  web:

    build: ./frontend

  api:

    build: ./backend

  db:

Now for our database we are not going to build a new image instead we will pull an image from docker – hub. So instead of *build* property we will use the ***image*** property.

version: "3.8"

services:

  web:

    build: ./frontend

  api:

    build: ./backend

  db:

    image: mongo:4.0-xenial

For db we use mongo version 4 build on top of *xenial* (*ubuntu version 16*).

Note: In Docker – hub we can see that mongo also has images built on top of windows. But windows images are very large (*over 2gb*). So linux images are preferred.

*For any of the services we can either build an image or pull it down*.

We also have port mappings by using ***ports*** property.

version: "3.8"

services:

  web:

    build: ./frontend

    ports:

**-** 3000:3000 🡪 *Map port 3000 of* ***host*** *to port 3000 of* ***container*** *running this image.*

  api:

    build: ./backend

    ports:

**-** 3001:3001

  db:

    image: mongo:4.0-xenial

Here we need to use array or list syntax. So we used a – (*hyphen*) and then define a port mapping.

Mongo db by default listens to port 27017 and we map the same to host port. So that we can access mongo db using a mongo db client like *mongo db compass*.

version: "3.8"

services:

  web:

    build: ./frontend

    ports:

      - 3000:3000

  api:

    build: ./backend

    ports:

      - 3001:3001

  db:

    image: mongo:4.0-xenial

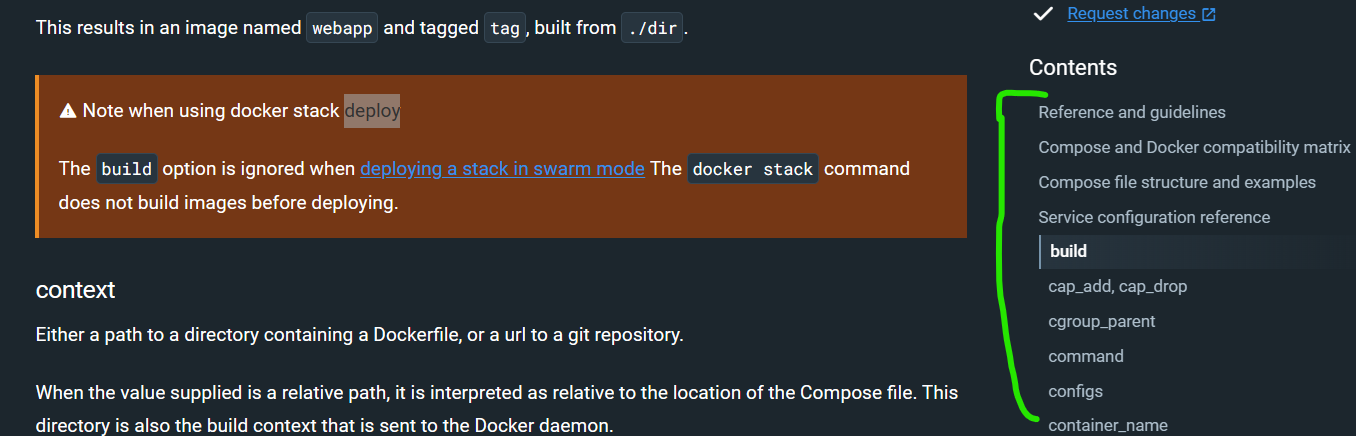
    ports:

      - 27017:27017

Note: If you don’t use mongo db, we have the same concept with other database engines. All these database engines listen on a default port and you need to map that port so that you can connect to the database engine using your favourite database client.

[Compose file version 3 reference | Docker Docs](https://docs.docker.com/compose/compose-file/compose-file-v3/#reference-and-guidelines)

On this page you can see various options that we can use in docker – compose version 3.



A lot of them are for really special cases and we don’t use them all the time. But the ones that we often use are ***build*** or ***image***. We also use ***ports***, ***volumes***, ***environment*** and so on…

Our api project needs an environment variable to tell where our database is. So we set environment to a list (*since we can have multiple environment variables*).

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      - DB\_URL=

Here we need to set *DB\_URL* to a *mongo db connection string*. These connection strings always starts with mongodb: followed by two forward slashes //. Then we type the name of our host.

Note: When we start an application with docker – compose, under the hood a network is created and on this network we are going to have 3 host. The names of these hosts are equal to the names that we have defined here (*web, api & db*).

So we are going to have a host called *db* which will be the connection string to our mongo db server in this case.

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      - DB\_URL=mongodb://db

On this server we can have multiple databases. So we will *specify the database name in the connection string as well*.

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      - DB\_URL=mongodb://db/vidly

This is one way to set an environment variable.

But instead of using list syntax we can also use *object* or *property – value* syntax.

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

In this syntax we get rid of hyphen and set the property and value for this property, we can add other environment variables along with it as well.

Next thing we need to add is ***volumes*** property, because we don’t want Mongo db to write data to the temporary file system of the container.

  db:

    image: mongo:4.0-xenial

    ports:

      - 27017:27017

    volumes:

      - vidly:

Here we set *volumes* to *vidly* (*name of our application, can be anything*) to a directory inside the container.

Note: If you look at the documentation of mongo db on docker hub or mongo db official documentation, you will see that by default mongo db stores its data in /data/db.

  db:

    image: mongo:4.0-xenial

    ports:

      - 27017:27017

    volumes:

      - vidly:/data/db

Hence we map this volume called vidly to this /data/db directory, so whatever is written inside this directory is actually outside this container (*somewhere else in our volume*).

Since we have used this volume here, we have to define it on our compose file. So we need to define another property called volumes and add another property called vidly with no value.

version: "3.8"

services:

  web:

    build: ./frontend

    ports:

      - 3000:3000

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

  db:

    image: mongo:4.0-xenial

    ports:

      - 27017:27017

    volumes:

      - vidly:/data/db

volumes: 🡪 *Here*

  vidly:

This looks a little weird but this is the syntax we have to follow.

*We just need to define a volume as a service first before we can use it*.

Let’s make our compose file, a little more readable by adding some line breaks in between.

version: "3.8"

services:

  web:

    build: ./frontend

    ports:

      - 3000:3000

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

  db:

    image: mongo:4.0-xenial

    ports:

      - 27017:27017

    volumes:

      - vidly:/data/db

volumes:

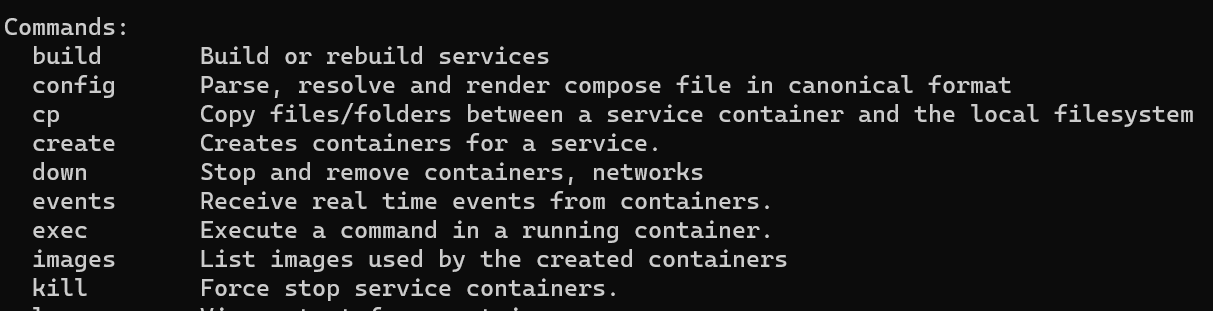
  vidly:

So we are done with our compose file. Next we will see how to build images.

**Building Images**:

As we know docker – compose is built on top of docker – engine. So *everything we have done with docker engine like building images, listing them, starting containers and so on… All these operations are also available using docker – compose*.

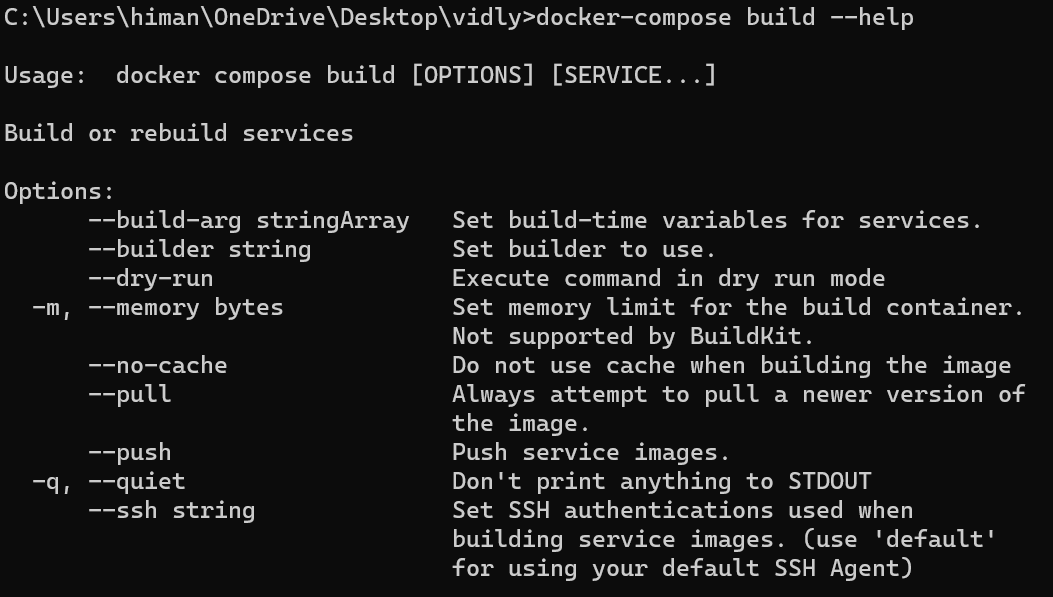
So write docker-compose without any arguments,



We have all these subcommands like rm, run, push, pull and so on… The difference is that any other commands will apply to our application as a whole.

So most of these commands will impact multiple services or multiple containers in our application.

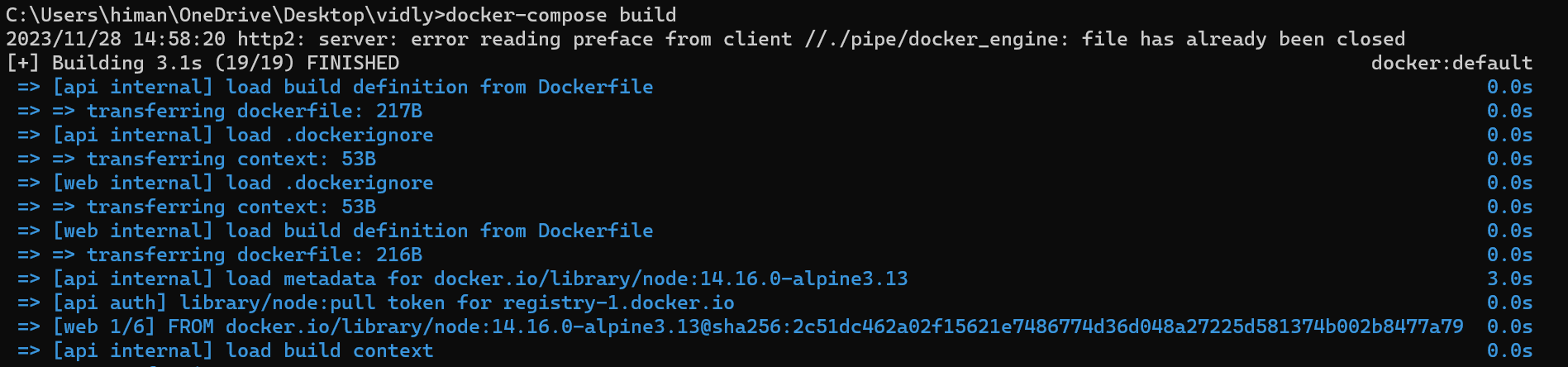
Let’s look at docker-compose build --help ,



We have a bunch of options here. Important ones are --no-cache (*to prevent caching when building images when we encounter weird issues*).

Another useful option is --pull (*with this we can always pull a newer version of the image*).

For now we are not going to use any of these options, we will simply docker-compose build.

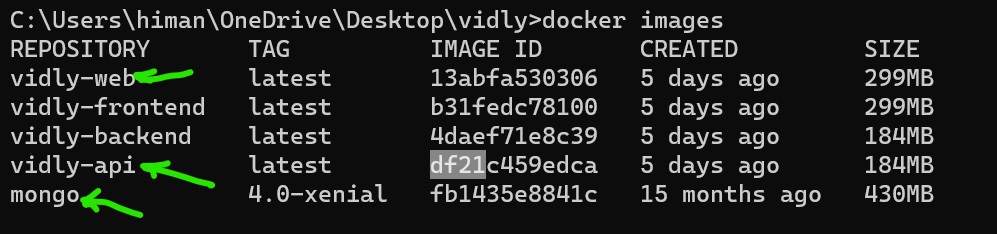


**Side NOTE**: Don’t know why I am getting this,

*2023/11/28 14:58:20 http2: server: error reading preface from client //./pipe/docker\_engine: file has already been closed*

Although image is created.

docker images



Now as part of this build process we built vidly – web& vidly – api (*while mongo we got from docker hub*).

The vidly-frontend and vidly-backend were created when we first started this application. Look at the original docker-compose.yml file we used earlier,

version: "3.8"

services:

  frontend: 🡪 *Here we used these names*

    depends\_on:

      - backend

    build: ./frontend

    ports:

      - 3000:3000

  backend: 🡪 *and this one*

    depends\_on: *instead of web and api*

      - db

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

    command: ./docker-entrypoint.sh

  db:

    image: mongo:4.0-xenial

    ports:

      - 27017:27017

    volumes:

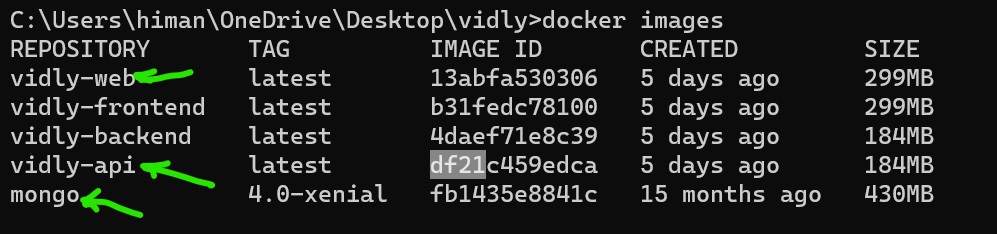
      - vidly:/data/db

volumes:

  vidly:

Note: When we build images using docker-compose our images are prefixed with the name of our application (*name of the directory*).

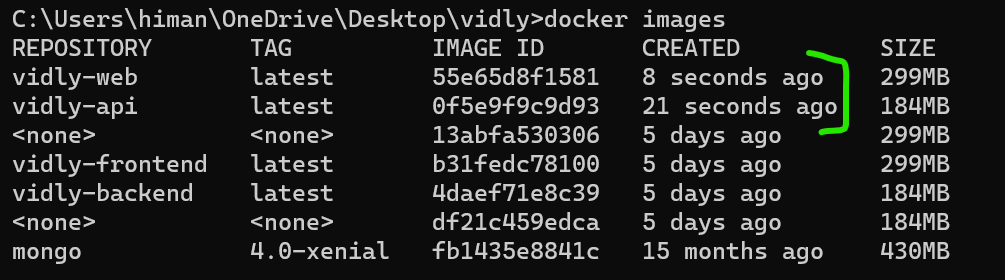
Note: If we look at the created column, we can see all these images were created 5 days ago even though we just built vidly-web and api images just now.



The reason is because vidly-frontend and backend images 5 days ago and now when we build new images docker used everything in cache, because all those files were already available (*all the layers already there*), so *docker did not had to do full rebuilt and we are still using the build from 5 days ago*.

If we want to force a full rebuild we can say,

docker-compose build --no-cache



Now we can see the images we just created.

**Starting and Stopping an Application**:

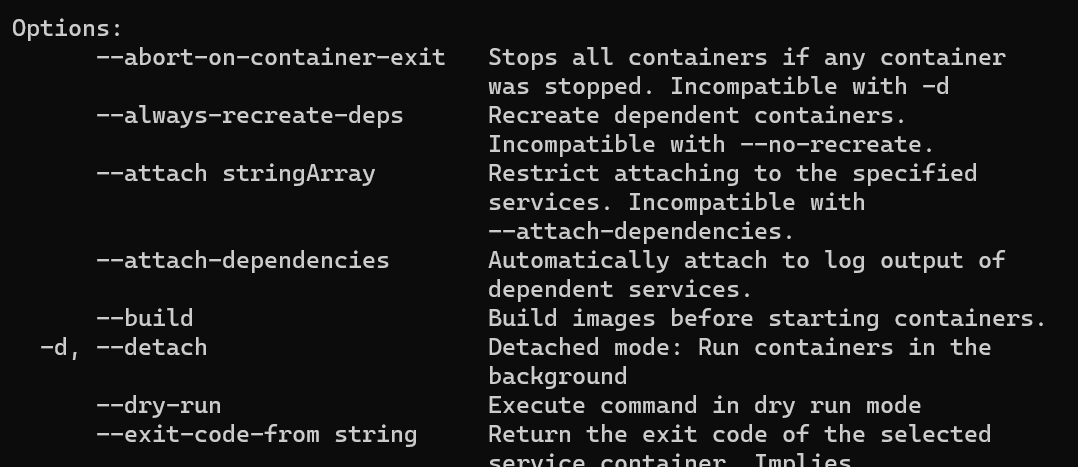
We briefly saw how can we start an application with docker – compose by using docker-compose up.

If the images are ready, docker-compose will run them inside containers, otherwise its going to build the images automatically.

Before running docker-compose up, let’s look at available options using,

docker-compose up --help

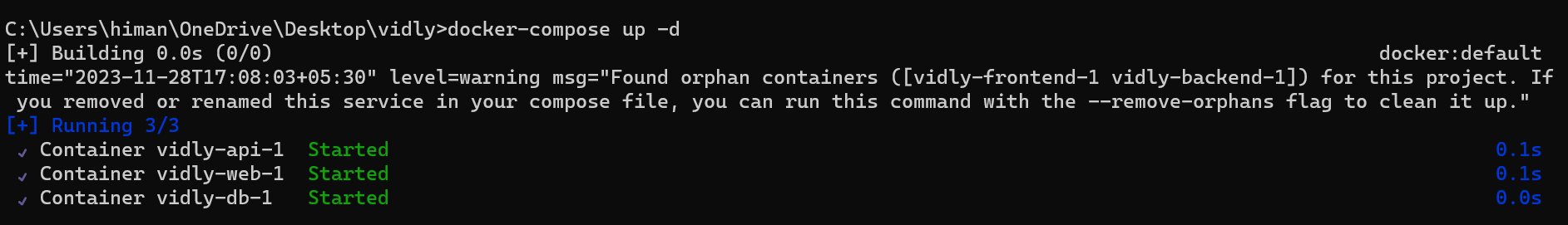
Here we have a ton of options, couple of them which are useful are.



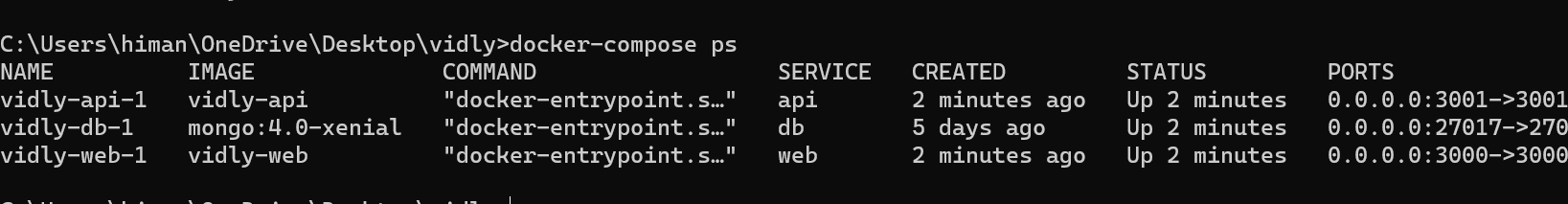
--build (*we can force rebuild every time we want to start our application*), so we don’t have to explicitly run docker-compose build and then docker-compose up. We can combine the two using **docker-compose up --build.**

The other useful option is -d for detached mode (*to start the containers in background*).

**docker-compose up -d**



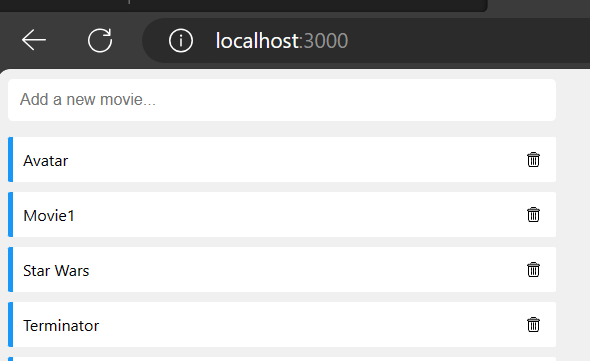
And now if we run **docker-compose ps**, we can see all the containers relevant to this application. In contrast if we run **docker ps**, we can see all the containers running across all applications.



Here we have three containers, vidly-api-1, vidly-db-1 and vidly-web-1. Now what is this -1 (*one*)?

Well we can start multiple containers from the same image and this is used for high availability and scalability.

Now if we go to port 3000, we can see our application.



Now how do we take this down. Let’s say *we are done with this application and we want to free up resources*. So back to the terminal we run **docker-compose down**.

*This will stop and remove these containers*.(*images are still there*)



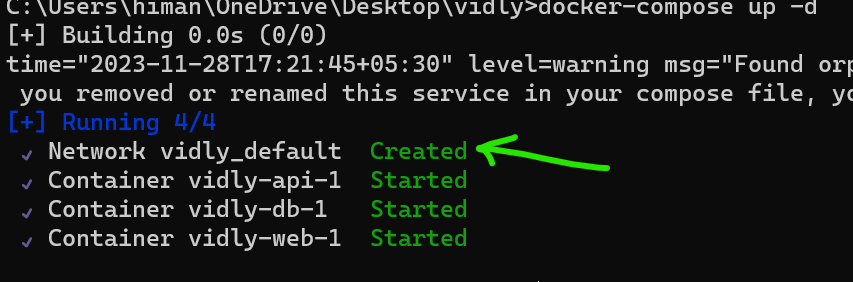
**Docker Networking**:

Let’s talk about networking in docker. *When we run our applications with docker-compose, it will automatically create a network and add our containers on that network*.

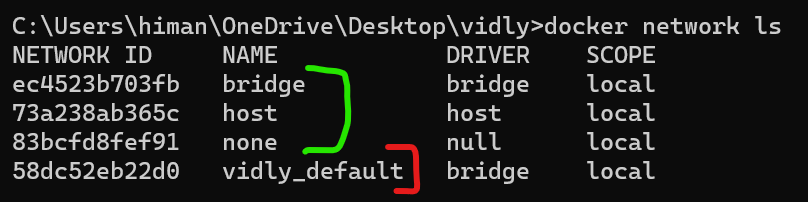
So these containers talk to each other. Let’s see this in action.

We will bring up our application one more time in detached mode.

docker-compose up -d



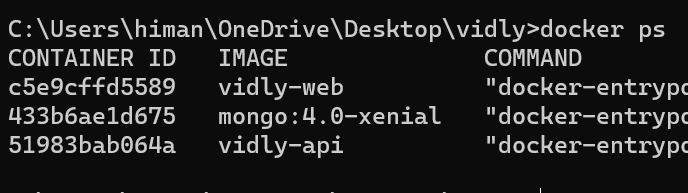
Look at the first line, Network vidly\_default. So we can run **docker network ls**



Here we can see all the networks on this machine. Every docker installation has three networks i.e. bridge, host and none. That’s not important for now what matters is vidly\_default.

This vidly\_default network contains three hosts or three containers (*web, api and db*). So these hosts or containers can talk to each other using their name.

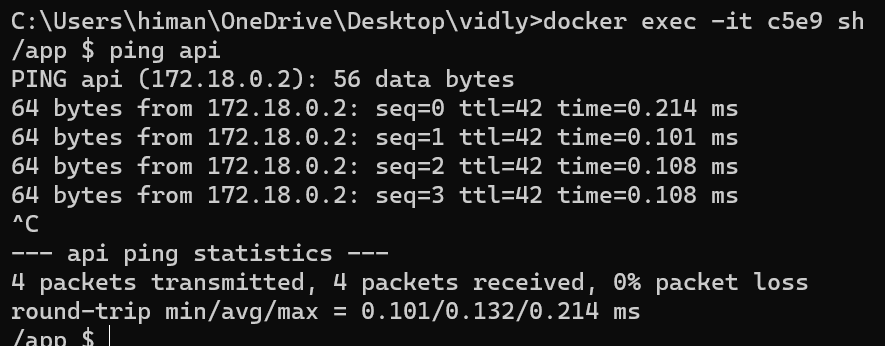
Back to the terminal, let’s look at the running containers.



Let’s start a shell session in the vidly-web container and ping the vidly-api container,

docker exec -it c5e9 sh

Then we will ping api (*here api is the name of api container*)

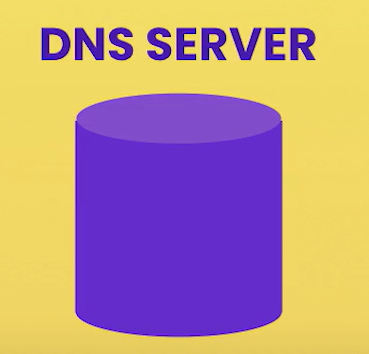


Side Note: If you get **ping: permission denied** error then it means we have not logged in as root user and ping operation requires root privileges. So with the last command we will use one extra *option for setting the user*.

docker exec -it -u root c5e9 sh

What happens under the hood?

Docker comes with an embedded DNS server that contains the name and IP of these containers.

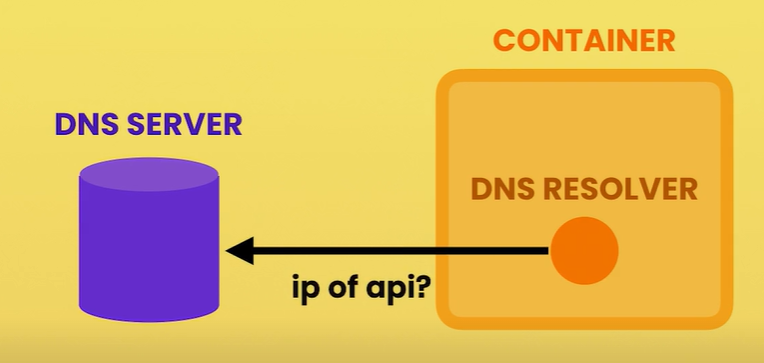


Now inside these containers, we have a component called the DNS resolver.

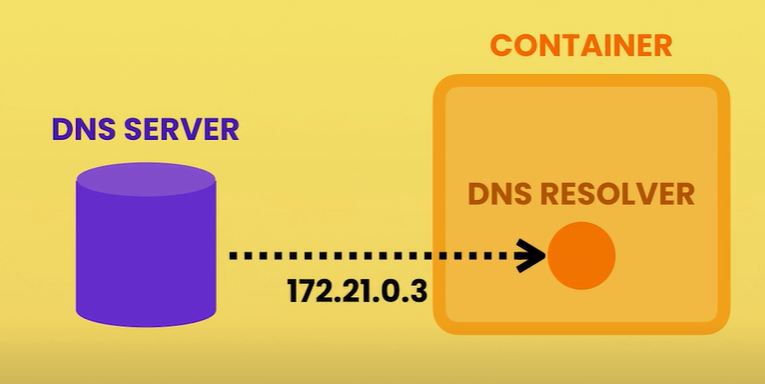


This *DNS resolver talks to the DNS server to find the IP address of the target container*.

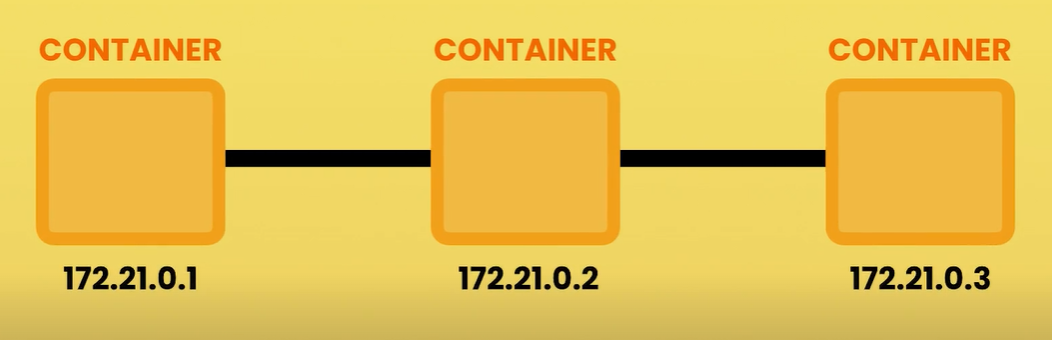
So when we ping the *api* container, this DNS resolver asks the server, what is the IP address of the api machine or api container.



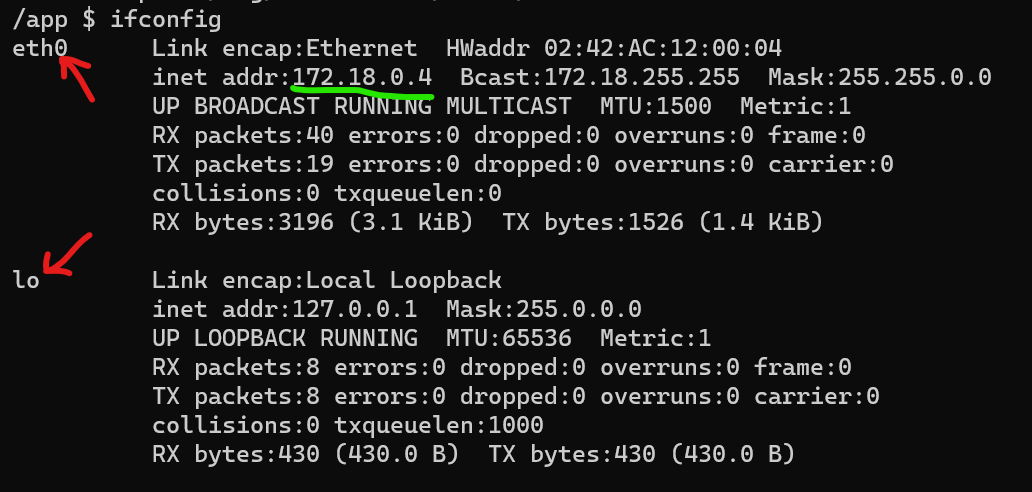
The DNS server returns the IP address and then web container can directly talk to api containers using its IP address.



So each container has an IP address and is part of a network.



If we type ifconfig inside shell session, we can see IP address of this container.



This container has two network adaptors. One of them is ethernet 0 or eth0 and we can see the IP address of this container (*172.18.0.4*).

Back to our compose file, earlier when we defined the api service, we added an environment variable that contains a database connection string.

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly 🡪 *Here*

In this string, we have db which is the name of host (*that is db host or db container*).

We know that our *api* container can talk to *db* container because both these containers or all containers of this application are part of the same network.

One thing we need to understand here is that this host (*db*) is only available inside the docker environment. So if we open up browser and go to *localhost/db*, we will not get anything.

So the api container can directly talk to the db container, but if we need to access this container, we need port mappings and that’s why we have this port mapping here.

  db:

    image: mongo:4.0-xenial

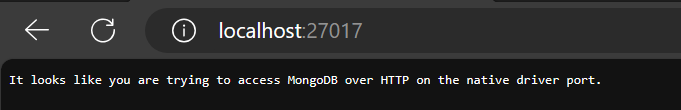
    ports:

      - 27017:27017 🡪 *Here*

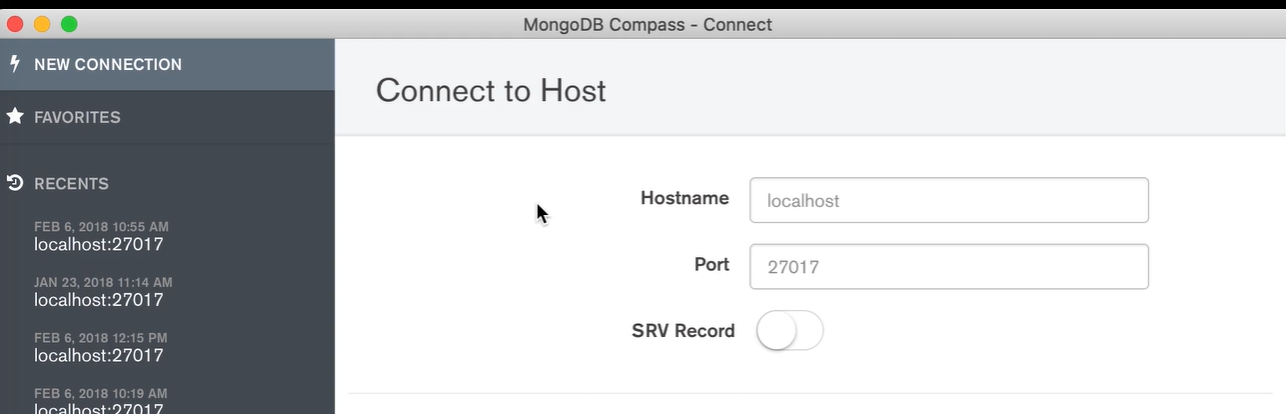
    volumes:

      - vidly:/data/db

We can connect with localhost:27017

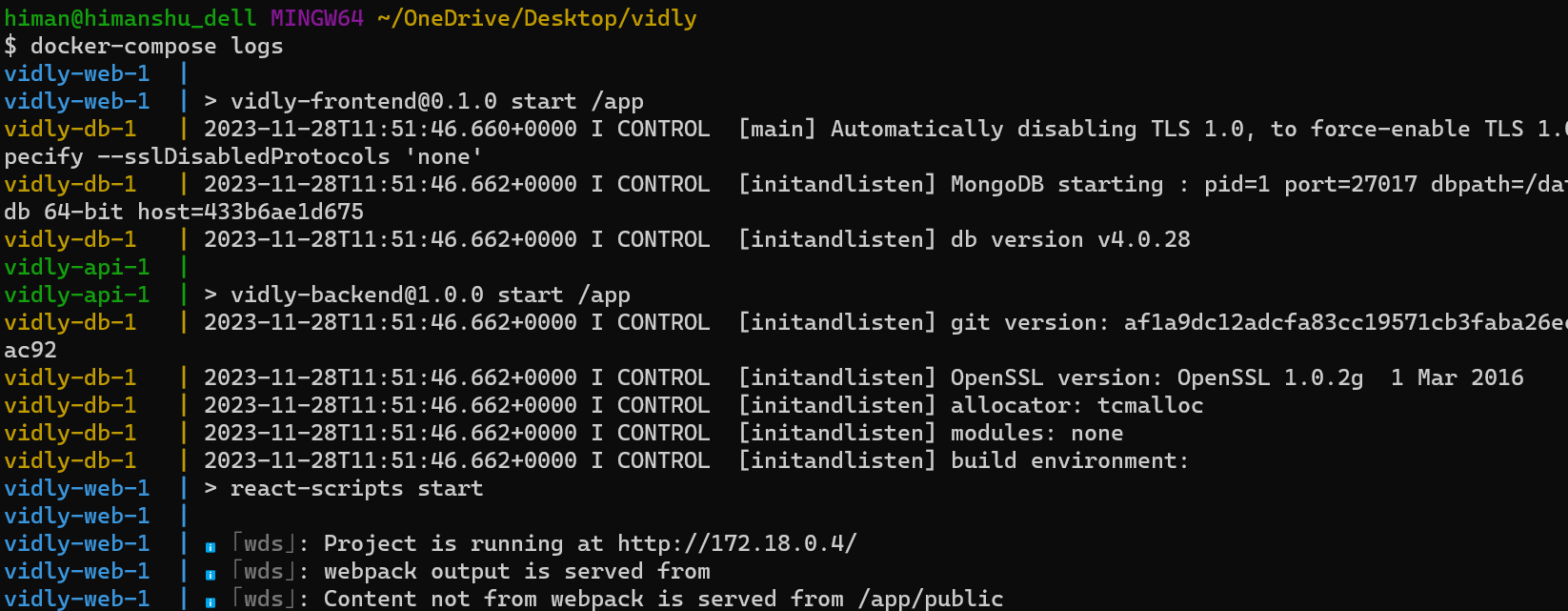


In mongodb compass we can connect using the same mappings.



**Viewing Logs**:

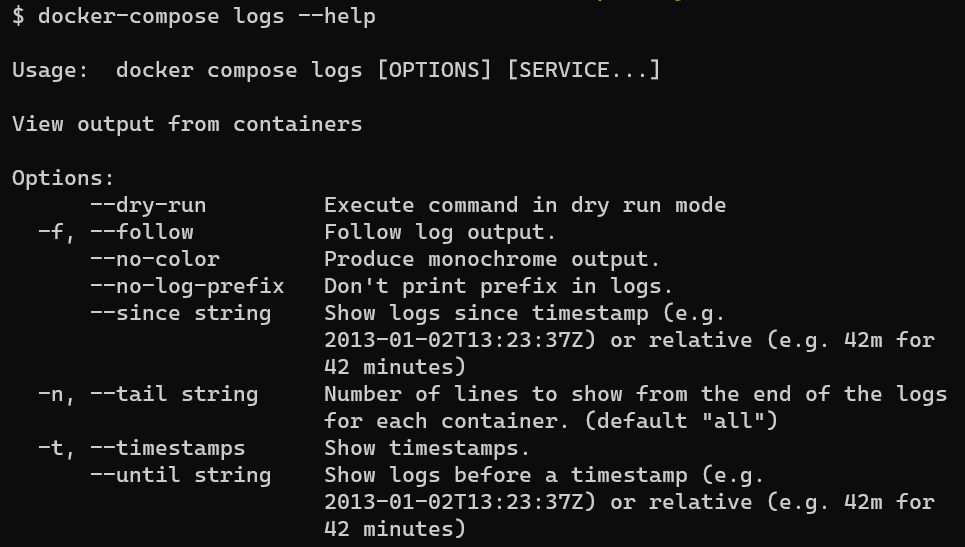
Let’s see a few different techniques to view the logs, if we type **docker-compose logs**, *we can view the logs across all containers of this application in one place*.



Here we have some messages coming from our database container, web container etc (*each one of them is colour coded*).

Let’s add --help at the end of this command to see all options available for logs command.

**docker-compose logs --help**



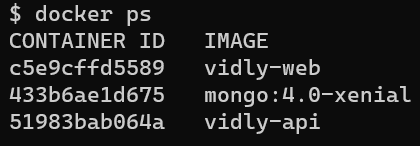
Here we have the same options that we learned about earlier in the course.

Using -f we can follow the log output, so that *we can continuously see new messages as they come out*.

We can *add a timestamp* using -t.

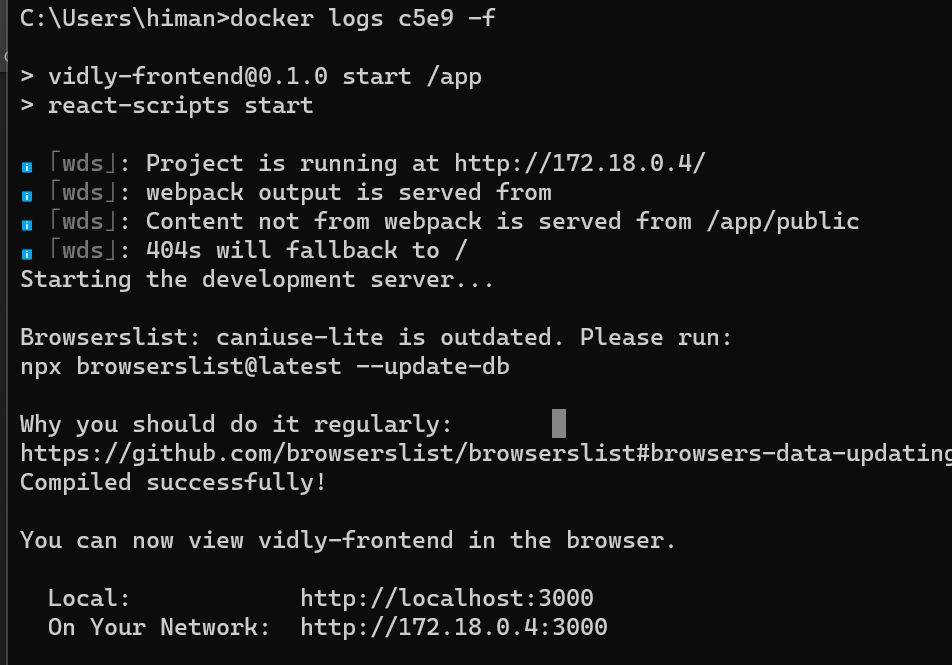
What if we don’t want to see logs for all these containers in one place? *In case we want to see logs of different containers on different monitor*s.

That’s very easy. We can look at the running containers.



Suppose we want to look at the logs of *web* container. We give the <container id> to docker logs command and if we want to follow the logs we supply -f option.

docker logs c5e9 -f



**Publishing Changes**:

Obviously *we don’t want to rebuild our images every time we change our code*. So we are going to map our project directory like the *backend* directory to the *app* directory inside our container.

This way any changes we make in this directory are immediately visible inside our container.

So let’s open up our compose file and here is the definition of our api service,

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

In this definition, we are going to add a new property called ***volumes***. Since we can have one or more volume mappings so here we can use list or array syntax.

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

    volumes:

      - ./backend:/app 🡪 *Here*

Here we typed a relative path. So starting from current directory we go to our backend directory and map this to app directory inside our container.

Note: In contrast, earlier, when we started our containers manually we had to type an absolute path like docker run -***v $(pwd)*** to map current working directory to the app directory.

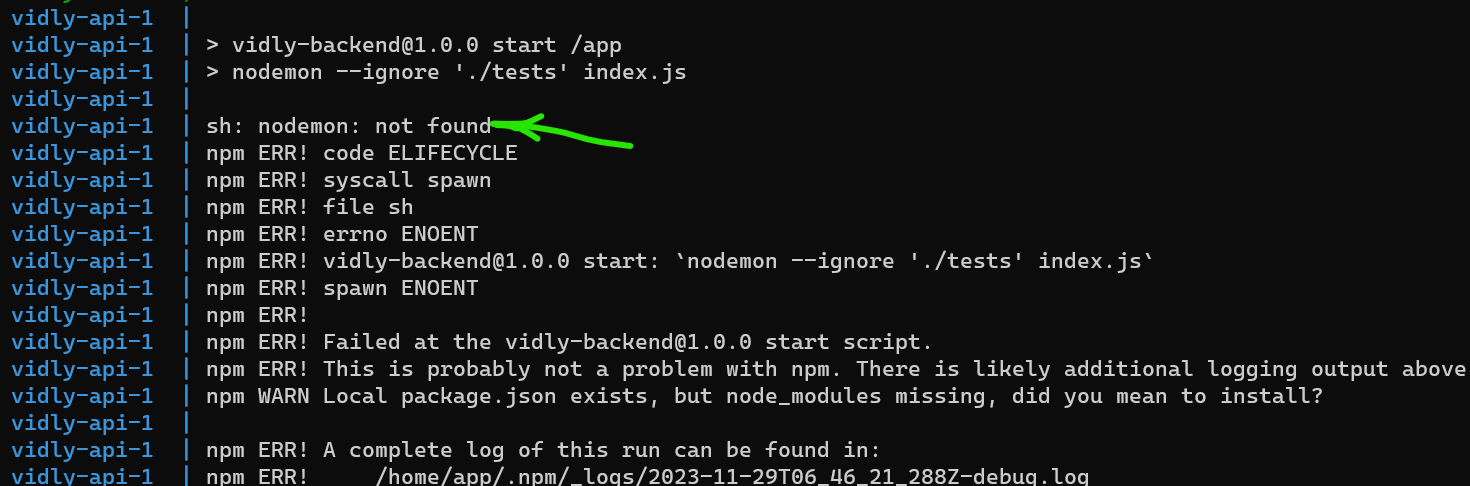
But with compose files, we don’t have to do this and we can type a relative path here which is lot easier.

But there is a problem with this implementation.

To see that let’s bring up our application,

docker-compose up

We see an error,



nodemon: not found, Here nodemon is one of the packages that our backend project is dependent on.

Go to our backend directory 🡪 package.json file 🡪 devDependencies:

  "devDependencies": {

    "jest": "^26.6.3",

    "migrate-mongo": "^8.1.4",

    "nodemon": "^2.0.7", 🡪 *Here*

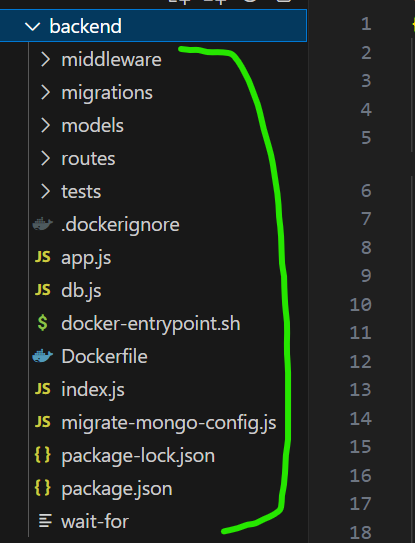
    "supertest": "^6.1.3"

  },

Here we have a dependency on nodemon or node monitor. It is a utility that watches all our files and anytime it detects a change, it restarts our node server.

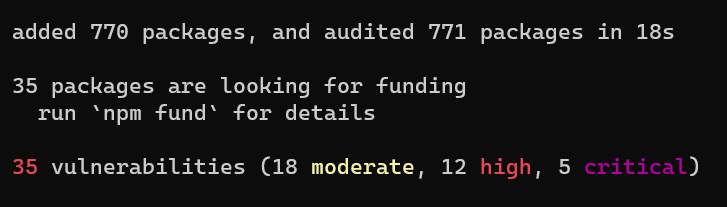
The reason is we do not have this dependency installed in our local machine. If we notice there is no node modules folder inside our backend directory (/vidly/backend) instead its installed inside docker container.

But since now we are sharing our applications code with our container, what the container sees is basically this directory.



And here we don’t have node modules folder.

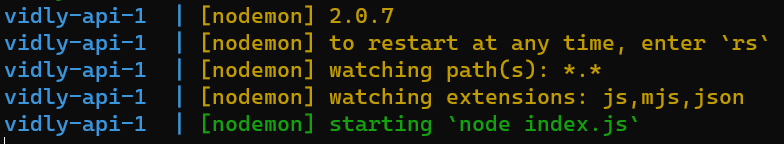
So let’s stop the container by ctrl + C, go to backend directory and then run npm install or npm i to install our dependencies.



All our dependencies are installed.

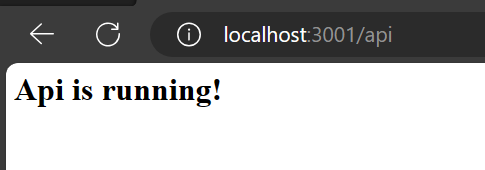
Now let’s bring up our application again.

docker-compose up



Here we can see nodemon starting our node process.

If we go to localhost:3001/api which is the home page of our API.



Now let’s make a small change in our code and see if that change is visible here or not.

So go to backend/routes/index.js and change our message from API is running! to API is running!!!!

router.get("/", (req, res) => {

  res.send("<h1>Api is running!!!!</h1>");

});

***ISSUE****: Changes are not reflecting on localhost:3001 after saving the file, nodemon should have restarted automatically*. Need to troubleshoot.

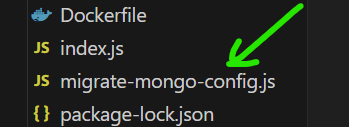
[Docker Compose not synchronising file changes in volume - Open Source Projects / Compose - Docker Community Forums](https://forums.docker.com/t/docker-compose-not-synchronising-file-changes-in-volume/79177/6)

This thread might help (*based on the discussion it might be windows issue but it can be overridden*).

**Migrating the database**:

Most of the time when we release our application, *we want our database to be in a particular shape with some data*, this is called *database migration*.

In this backend project, we are using a database migration tool called migrate-mongo.



And if we look at package.json file,

  "devDependencies": {

    "jest": "^26.6.3",

    "migrate-mongo": "^8.1.4", 🡪 *We have dependency on this package*

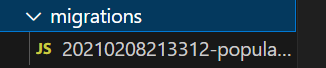
    "nodemon": "^2.0.7",

    "supertest": "^6.1.3"

  },

We have similar tool in other development stacks for example in dot net we have entity framework and in Django we also have database migrations.

Using these database migration tools we can create database migration scripts. In this project our migration scripts are stored in this migrations folder.

**

Currently here we have one script which is just a general JavaScript file.

module.exports = {

  async up(db, client) {

    await db

      .collection("movies")

      .insertMany([

        { title: "Avatar" },

        { title: "Star Wars" },

        { title: "Terminator" },

        { title: "Titanic" },

      ]);

  },

  async down(db, client) {

    await db.collection("movies").deleteMany({

      title: {

        $in: ["Avatar", "Star Wars", "Terminator", "Titanic"],

      },

    });

  },

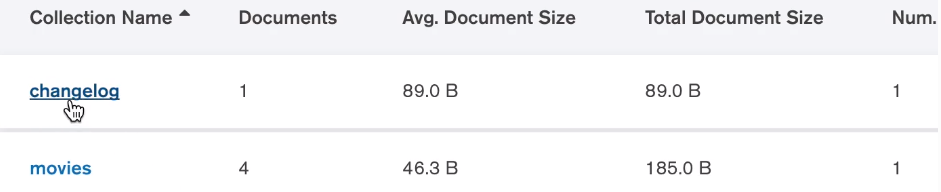
};

In each file we have two functions up (*for upgrading the database*) and down (*for downgrading it*).

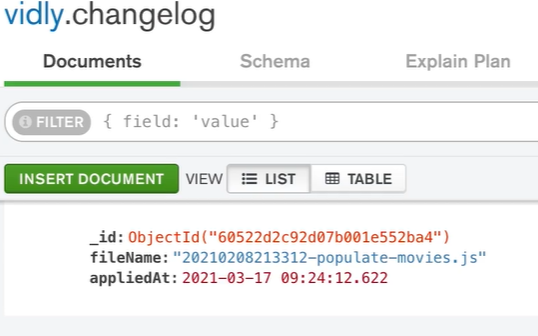
In the up function we are going to the movies collection and inserting given movies objects and when downgrading we are removing the same ones.

Now using a tool like migrate-mongo we can go to the terminal and say migrate-mongo up (*this will look at our migrations folder and execute all migration scripts*).

What if we run this command multiple times? Our scripts are not going to be executed twice, because if we look at our database. Here we have a collection called changelog.



And in this collection, we have the scripts that have been executed.



So migrate-mongo will not execute the script multiple times.

Back to package.json, in the *scripts* section we have these commands which are aliases to the command in front of them.

  "scripts": {

    "db:up": "migrate-mongo up",

    "start": "nodemon --ignore './tests' index.js",

    "test": "jest --watchAll --colors"

  },

Here we have db:up which is mapped to migrate-mongo up.

So in the terminal we can either say

migrate-mongo up or npm run db:up

With npm run we can run any of the commands in package.json file.

Now how can execute a database migration as part of starting our application? For this let’s look at Dockerfile in our backend project.

FROM node:14.16.0-alpine3.13

RUN addgroup app && adduser -S -G app app

USER app

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3001

CMD ["npm", "start"]

Here in the CMD instruction we are simply calling npm start which starts our webserver. *In our compose file we can override the command in Dockerfile and do something else*.

So in the definition of our api service,

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

    volumes:

      - ./backend:/app

We will add a new property here called ***command*** and here we can override the default command defined in our Dockerfile.

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

    volumes:

      - ./backend:/app

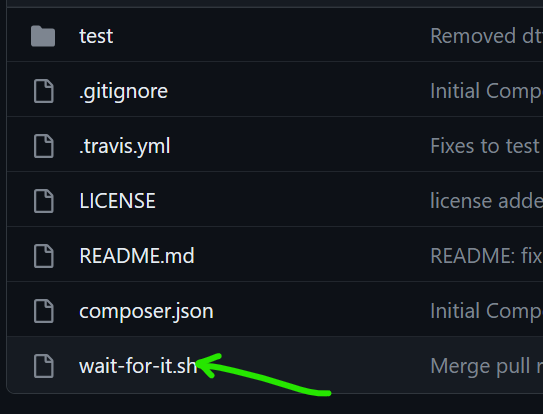
    command: migrate-mongo up && npm start 🡪 *Here*

But there is a problem here, it is possible that our database server is not ready at the time of executing this command(*migrate-mongo up*) so *even though our database container might be running the actual database engine (mongo:4.0-xenial) may not be ready. Its because starting a database engine often takes several seconds*.

This is where we use a waiting script. There are many tools for this purpose but the one we going to use here is **wait-for-it**.

[vishnubob/wait-for-it: Pure bash script to test and wait on the availability of a TCP host and port (github.com)](https://github.com/vishnubob/wait-for-it)

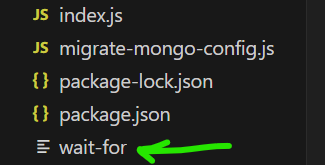
This is a GitHub repository which includes a shell script.



With this script we can wait for our database engine or any other processes to get ready before doing some work.

So in our compose file, before migrating our database, we want to use our wait-for-it script.

In our backend project folder we have stored our wait-for script here.



So here we can say *./wait-for* then we can specify the name of our computer or host which is *db* and then we want to wait for port *27017* to receive traffic (*default MongoDB port*).

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

    volumes:

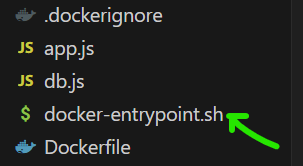
      - ./backend:/app

    command: ./wait-for db:27017 && migrate-mongo up && npm start

Once this is ready then we migrate-mongo.

As we can see this command is a little bit too long so there is another way to achieve the same result which is by creating an *entrypoint script*.

In the same folder we have created another file called docker-entrypoint.sh



This is a basic shell script where first we echo a message saying “waiting for MongoDB to start” and here we use the wait-for script.

#!/bin/sh

echo "Waiting for MongoDB to start..."

./wait-for db:27017

echo "Migrating the databse..."

npm run db:up

echo "Starting the server..."

npm start

Now the MongoDB is up and running, we print a message saying “Migrating the database” and run npm run db:up

And Finally we start our webserver.

Now we can go to our compose file and simplify our ***command***.

  api:

    build: ./backend

    ports:

      - 3001:3001

    environment:

      DB\_URL: mongodb://db/vidly

    volumes:

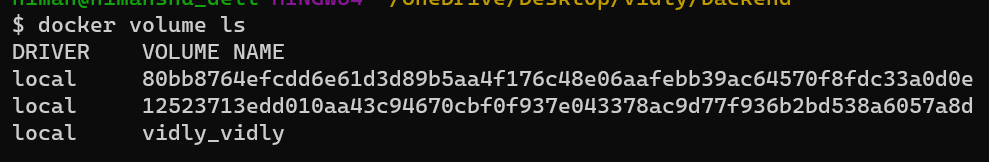
      - ./backend:/app

    command: ./docker-entrypoint.sh

This is for linux, in windows a PowerShell script can be written for same purpose.

To verify if this is working, go to terminal , stop the process and restart. But first look at the volumes.

docker volume ls



One of the volume is *vidly\_vidly*. First vidly represents our application and second vidly is the name of our volume.

Let’s remove this volume, so that our MongoDB database is gone.