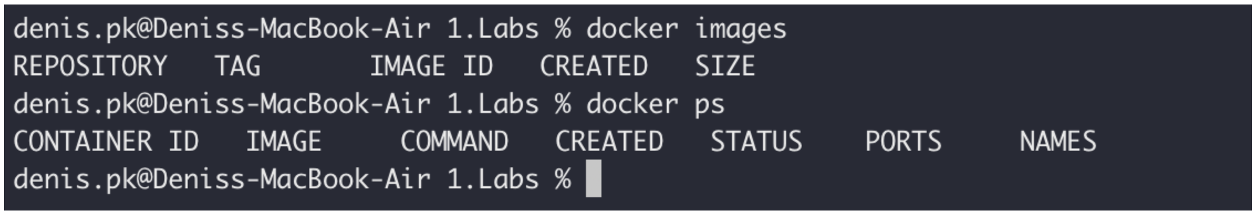
# Lab 3

## Denis Perepelyuk C00259076

## Containerize an Application

To show that we are working from a clean slate and have no other images or containers on our machine



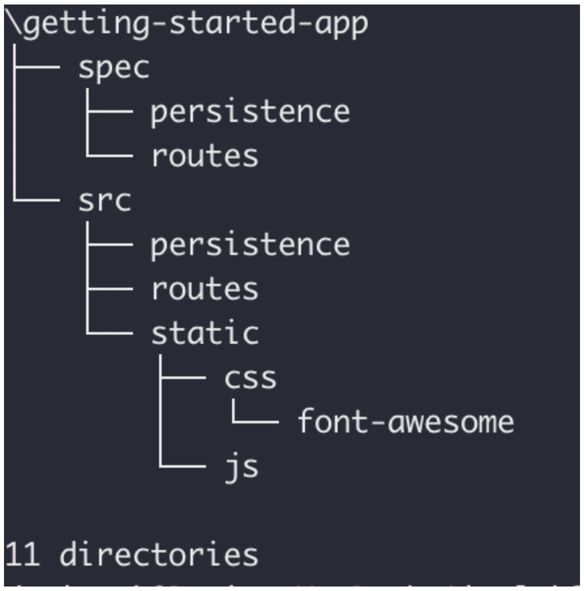
```
denis.pk@Deniss-MacBook-Air 1.Labs % docker images
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
denis.pk@Deniss-MacBook-Air 1.Labs % docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
denis.pk@Deniss-MacBook-Air 1.Labs % 
```

We can clone the repo by running the following command:

```
git clone https://github.com/docker/getting-started-app.git
```

By running the following code, we can see the directory structure:

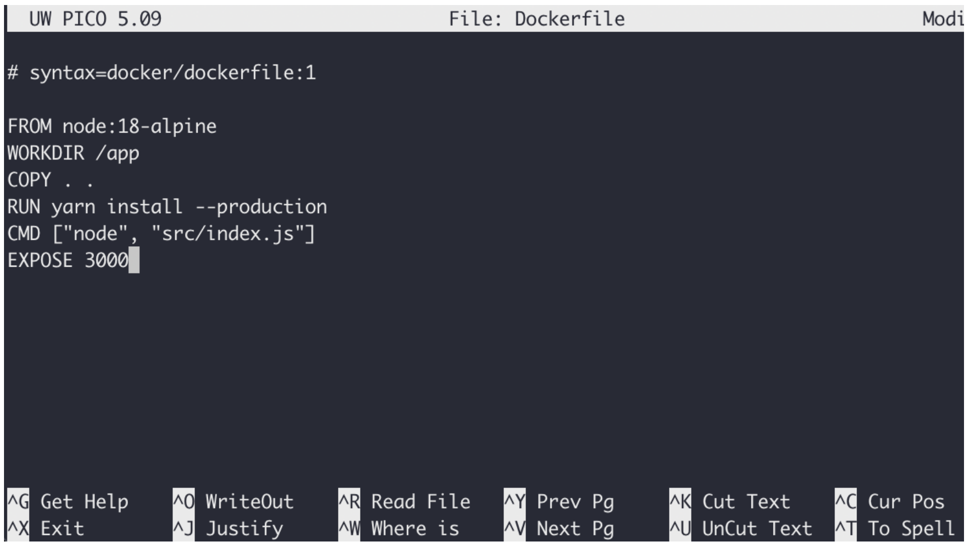
```
tree -d getting-started-app
```



```
\getting-started-app
├── spec
│   ├── persistence
│   └── routes
└── src
    ├── persistence
    ├── routes
    └── static
        ├── css
        │   └── font-awesome
        └── js

11 directories
```

After we have cloned the repo we have to create a file called “Dockerfile” with the following contents.

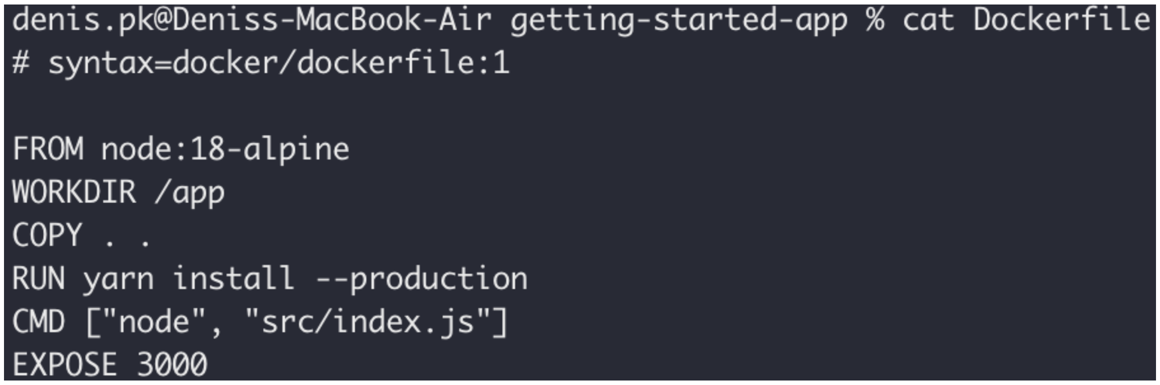


```
  UW PICO 5.09                     File: Dockerfile                      Modi

# syntax=docker/dockerfile:1

FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

To ensure that our changes have been saved we can use the **cat** command to view its contents.


```
denis.pk@Deniss-MacBook-Air getting-started-app % cat Dockerfile
# syntax=docker/dockerfile:1

FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

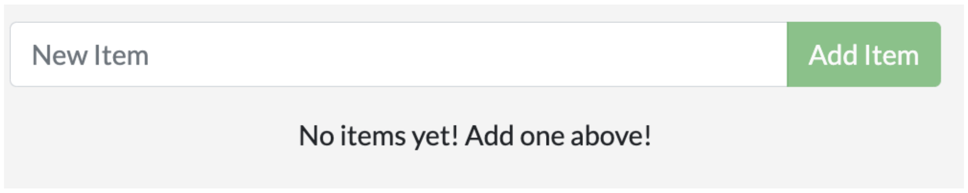
By running the following command we can build our image:

```
docker build -t getting-started .
```

We can run our app using the following command:

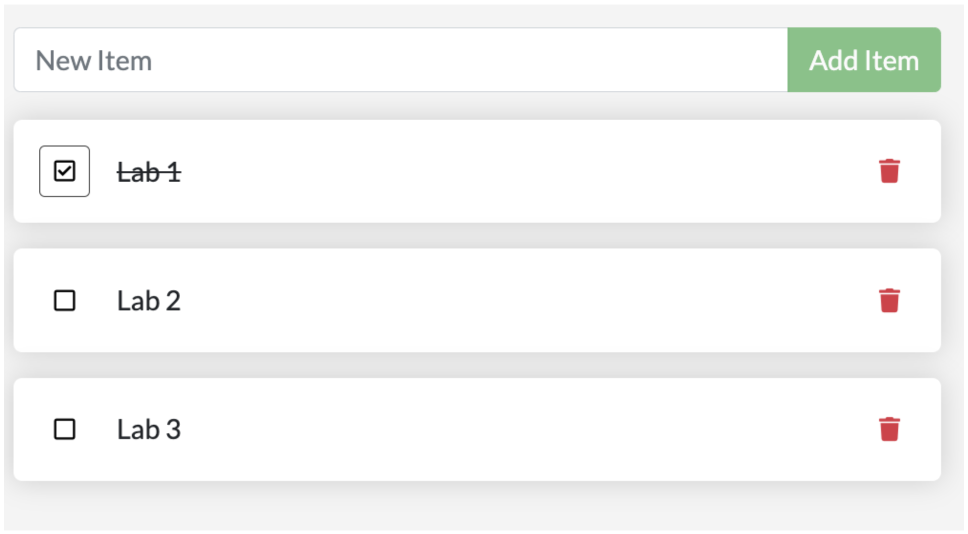
```
docker run -dp 127.0.0.1:3000:3000 getting-started
```

An by accessing http://localhost:3000/, we get the following:

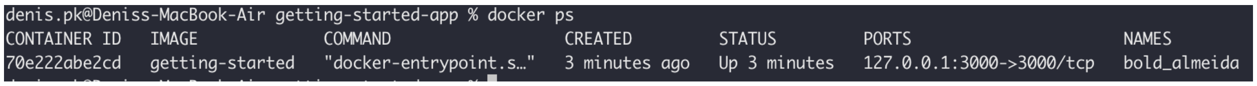


No items yet! Add one above!

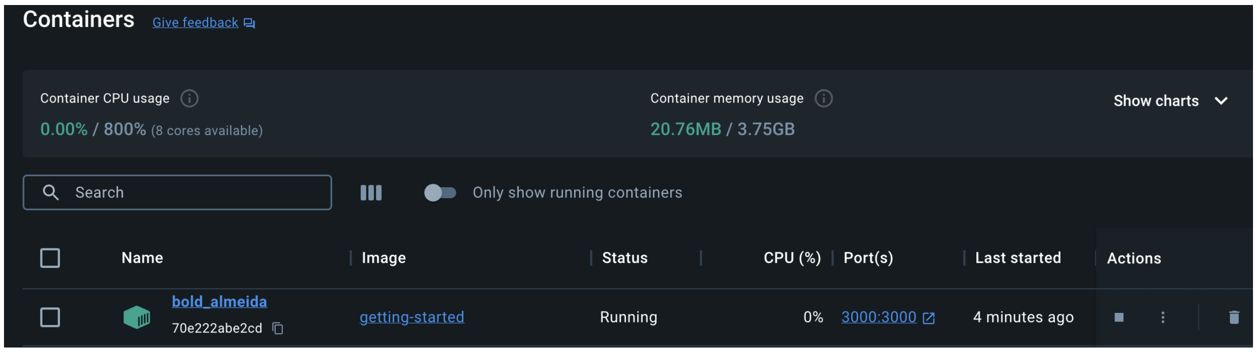
An our app is fully functional as can be seen below:



We can see from the terminal that our app is fully running as well as from the Docker Desktop application.



```
denis.pk@Deniss-MacBook-Air getting-started-app % docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED         STATUS         PORTS                      NAMES
70e222abe2cd   getting-started   "docker-entrypoint.s…"   3 minutes ago   Up 3 minutes   127.0.0.1:3000->3000/tcp   bold_almeida
```



Containers  Give feedback

Container CPU usage 0.00% / 800% (8 cores available)
Container memory usage 20.76MB / 3.75GB
Show charts

Search  Only show running containers

| Name | Image | Status | CPU (%) | Port(s) | Last started | Actions |
|---|---|---|---|---|---|---|
| bold_almeida 70e222abe2cd | getting-started | Running | 0% | 3000:3000 | 4 minutes ago | |

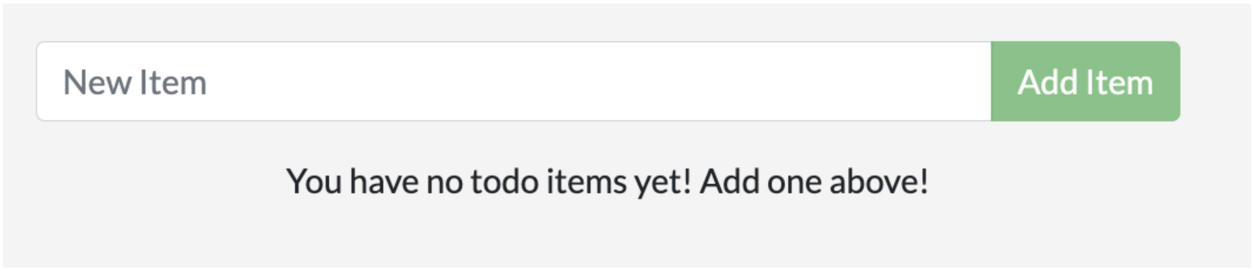
## Update the Application

Lets modify the file located at: **src/static/js/app.js** and change line 56 to the following:

```
- <p className="text-center">No items yet! Add one above!</p>
+ <p className="text-center">You have no todo items yet! Add one
```

We can build our updated version and run the updated app using the following command:

```
docker build -t getting-started .
docker run -dp 127.0.0.1:3000:3000 getting-started
```



You have no todo items yet! Add one above!

## Persist the DB

Now we will make it so that the list does not appear as empty every time we launch the container.

In Docker, containers have their own "scratch space" which is used to be able to create, update and remove files. These changes are not visible in other containers using the same image, as if each container were it's own separate entity from each other.

We can see this by starting two containers. Within one container we will create a file and the second will be used to verify the existence of the file. We should see that it doesn't.

Here we will create a container running Linux Alpine and run some commands through it's shell.

```
docker run -ti --name=mytest alpine
```

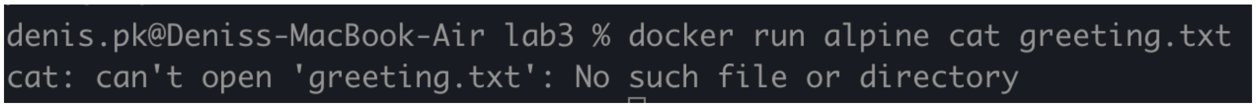
We will run the following commands to create a file called "greeting.txt" that will store the text "hello". Once this file is created we will exit the shell.

```
echo "hello" > greeting.txt
exit
```

Now we will run a new Alpine container and use the **cat** command to verify the existence of our "greeting.txt" file.

```
docker run alpine cat greeting.txt
```

As we can see, in the second container this file does not exist even though we are using the same image.



```
denis.pk@Deniss-MacBook-Air lab3 % docker run alpine cat greeting.txt
cat: can't open 'greeting.txt': No such file or directory
```

## Volumes

We will now remove these two containers. We now see that each container will start from the image definition and even if we make changes to the files within a container, these changes will be lost the next time we run a container with the same image. However, this can be avoided by using **volumes**.

To see volumes being used in practice, we will now create a one and start a container.

```
docker volume create todo-db
```

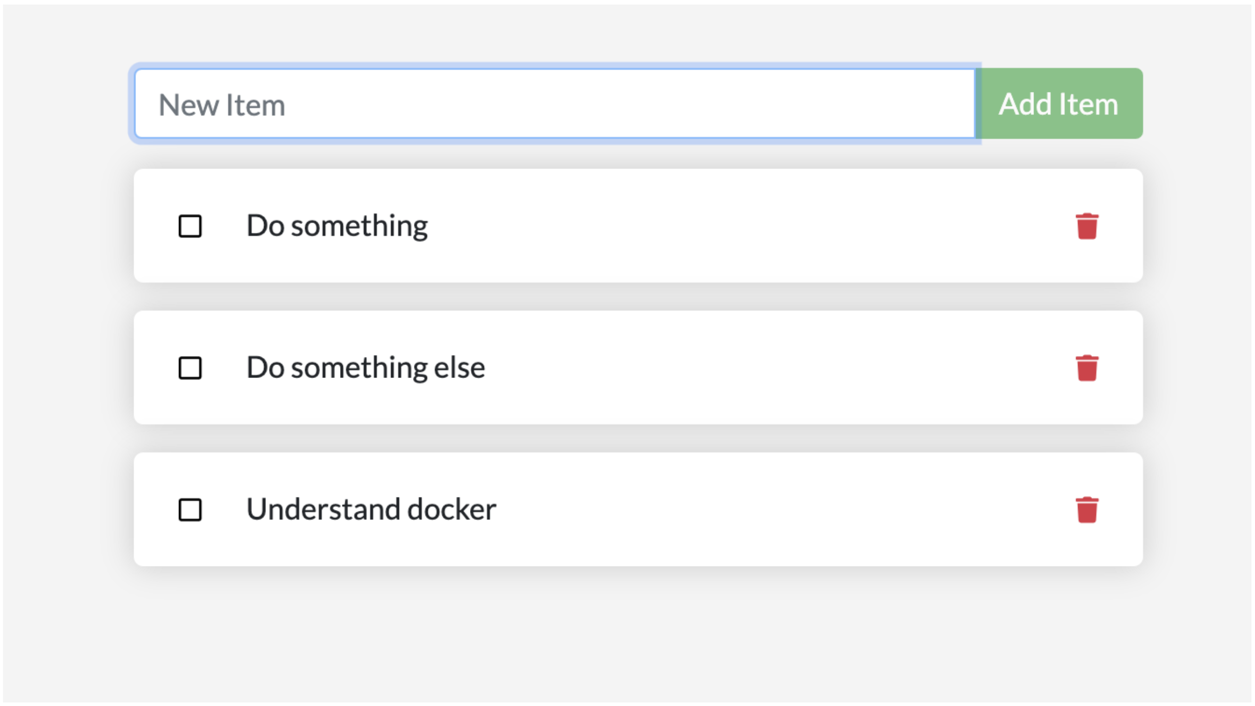
We will now stop and remove our todo container that we have been using previously.

```
docker rm -f b5757977b958
```

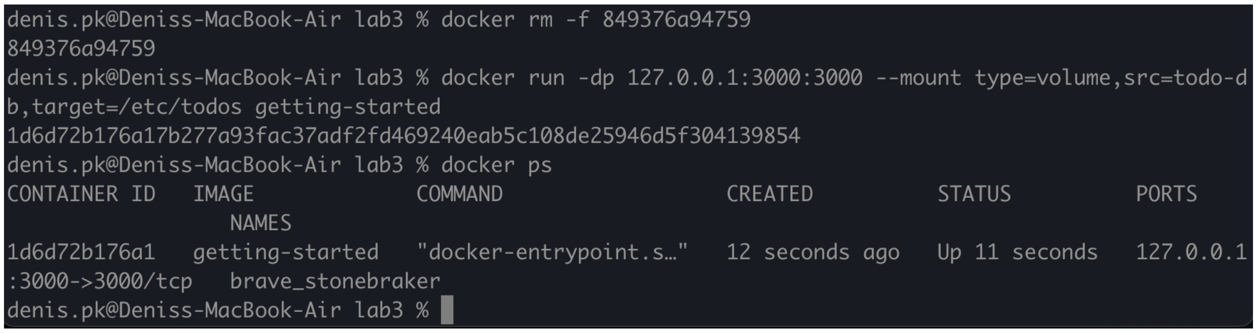
We will start our todo app container again, however this time we will use the **— mount** wildcard in order to specify a volume mount.

```
docker run -dp 127.0.0.1:3000:3000 --mount type=volume,src=todo
```

Now that the app is running once again, we can add some items to our todo list.

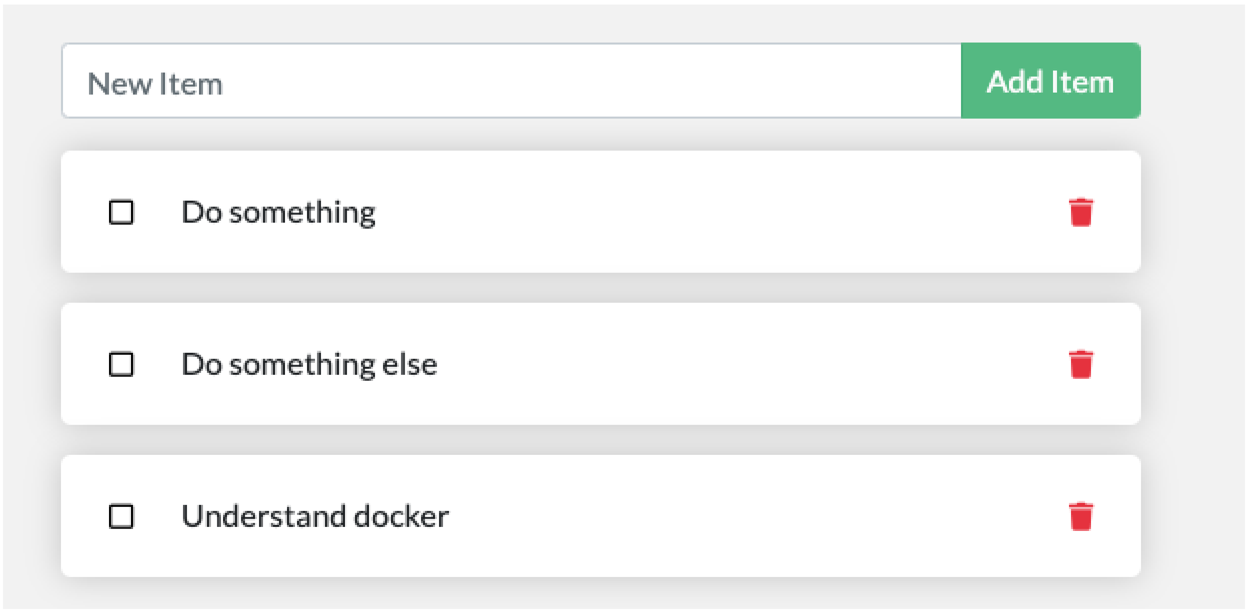


As done previously, we will remove the container for our app and start a new container following the previous steps.



```
denis.pk@Deniss-MacBook-Air lab3 % docker rm -f 849376a94759
849376a94759
denis.pk@Deniss-MacBook-Air lab3 % docker run -dp 127.0.0.1:3000:3000 --mount type=volume,src=todo-d
b,target=/etc/todos getting-started
1d6d72b176a17b277a93fac37adf2fd469240eab5c108de25946d5f304139854
denis.pk@Deniss-MacBook-Air lab3 % docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS          PORTS
                NAMES
1d6d72b176a1   getting-started   "docker-entrypoint.s…"   12 seconds ago   Up 11 seconds   127.0.0.1
:3000->3000/tcp   brave_stonebraker
denis.pk@Deniss-MacBook-Air lab3 %
```

As we can see, our previous items added to the todo list are still there even though we have removed the container that we used to add them and are using a completely different container.



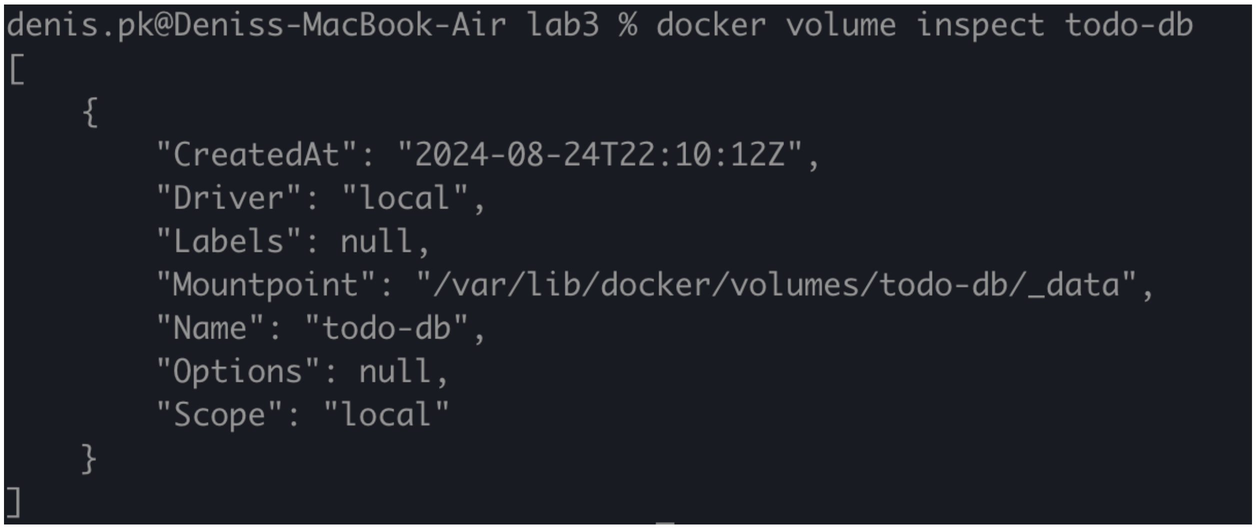
This is how we can persist data within Docker containers.

## Diving into the volume

If we want to see where exactly Docker is storing our data when we use a volume, we can do so by running the following command.

```
docker volume inspect todo-db
```

This is the output given in JSON format showing us where and how our data is being stored. “Mountpoint” refers to the actual location of our data on the disk.



```
denis.pk@Deniss-MacBook-Air lab3 % docker volume inspect todo-db
[
    {
        "CreatedAt": "2024-08-24T22:10:12Z",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/todo-db/_data",
        "Name": "todo-db",
        "Options": null,
        "Scope": "local"
    }
]
```

We now have learned how to be able to persist data within our images that can be stored and accessed by other containers.

## Use Bind Mounts

In the previous section we learned about volume mounts. In this section we will learn about **bind mount** which can be used to share a directory from the file system of the host into a container. An example of how we could use this is mounting source code from the local file system into a container and have changes made to the code be seen in the container immediately.

We will first run a quick experiment so that we achieve an understanding of how bind mounts work.  We will first start a container to run bash with a bind mount. The **—mount** option is used to tell Docker to create a bind mount, while **src** is used

```
docker run -it --mount type=bind,src="$(pwd)",target=/src ubunt
```