<https://www.bmc.com/blogs/mongodb-docker-container/>

How To Run MongoDB as a Docker Container

MongoDB is among the most popular [NoSQL databases](https://www.bmc.com/blogs/sql-vs-nosql/" \t "_self) today. And containers offer easy app usage and scalability. In this article, I’ll show you how to:

* Configure MongoDB as a container in Docker
* Set up the Docker platform with docker-compose
* Create a docker-compose file to create the MongoDB container
* And more

The last part of this tutorial will look at advanced configurations. These can give you a glimpse of the extensibility of a containerized project. So, we’ll create a self-containing project with a MongoDB instance and Mongo Express web interface on a dedicated network and docker volume to maximize the portability of the project.

Let’s get started.

*(This article is part of our [MongoDB Guide](https://www.bmc.com/blogs/mongodb-overview-getting-started-with-mongodb/" \t "_self). Use the right-hand menu to navigate.)*

Docker containers & MongoDB

[Docker](https://www.bmc.com/blogs/docker-101-introduction/) is a tool to create, deploy, and run applications using containers easily. A [container](https://www.bmc.com/blogs/what-is-a-container-containerization-explained/) is a standard unit of software that can be used to package applications and all the dependencies to a single package. These containers can be run on any server platform regardless of the underlying configuration or hardware structure.

Docker can be used to run MongoDB instances. Setting up MongoDB as a container allows the user to create a portable and extensible NoSQL database. A containerized MongoDB instance behaves exactly like a non-containerized MongoDB instance without having to worry about the underlying configuration.

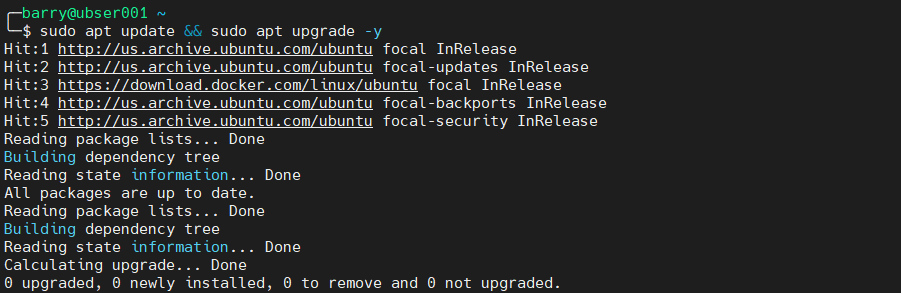
Installing Docker

In this section, we’ll set up a simple Docker installation to run containers on a Ubuntu-based server. We can get the Docker installation packages from [the official](https://docs.docker.com/docker-hub/repos/) Docker repository. Here are the installation steps:

1. Update existing packages.

Copy

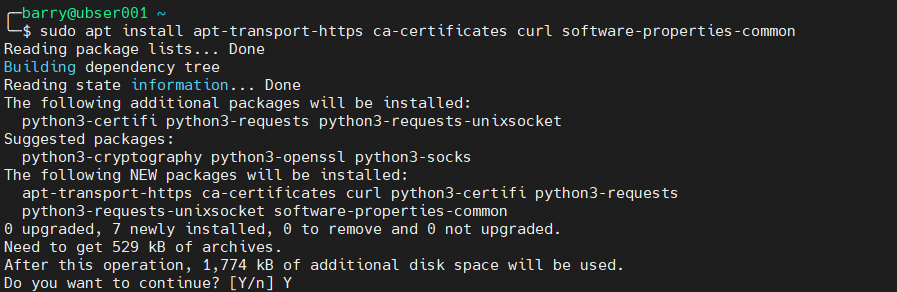
sudo apt update && sudo apt upgrade -y



1. Install prerequisite packages.

Copy

sudo apt install apt-transport-https ca-certificates curl software-properties-common



1. Add the GPG key from the official Docker repository.

Copy

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

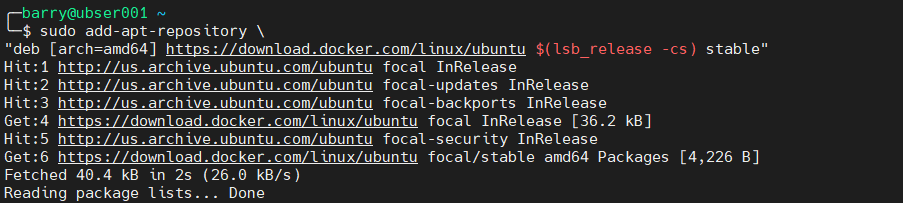
https://s7280.pcdn.co/wp-content/uploads/2020/12/key-80.png

1. Add the official docker repository to APT sources.

Copy

sudo add-apt-repository \

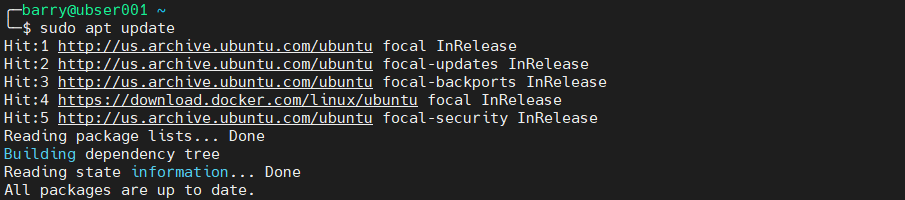
"deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"



1. Update the Ubuntu package list.

Copy

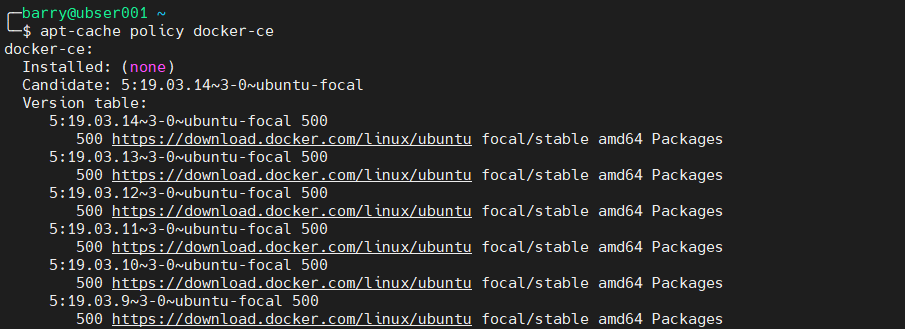
sudo apt update



1. Verify the docker repository.

Copy

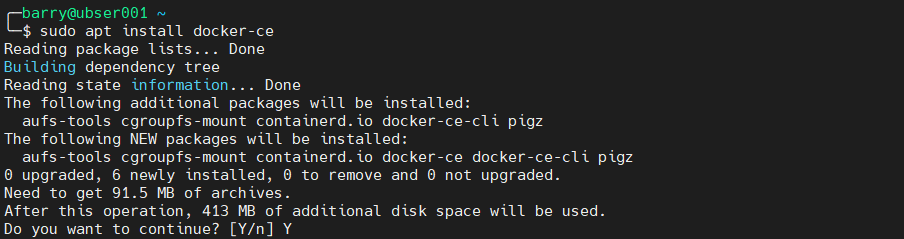
apt-cache policy docker-ce



1. Install the Docker community edition.

Copy

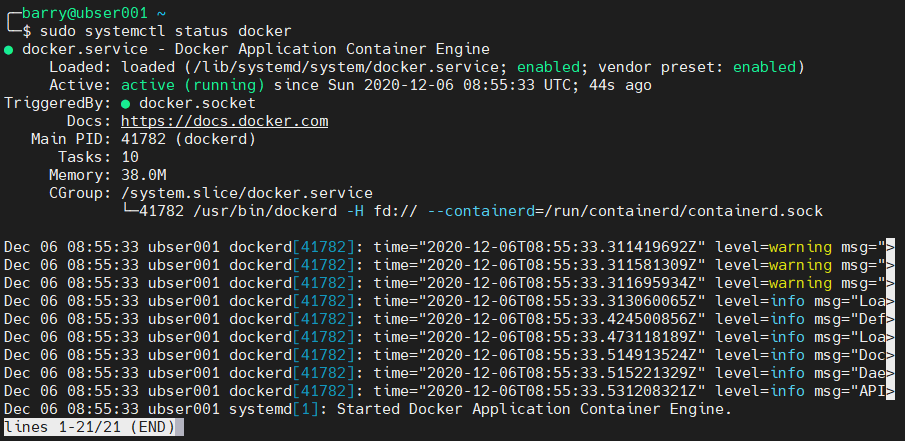
sudo apt install docker-ce



1. Check the status of the installation with the following command. If the **service** status returns active (running), Docker is successfully installed and active on the system.

Copy

sudo systemctl status docker



Installing Docker Compose

We can use the command line interface (CLI) to create and manage Docker containers. However, the CLI can be tedious when dealing with multiple containers and configurations.

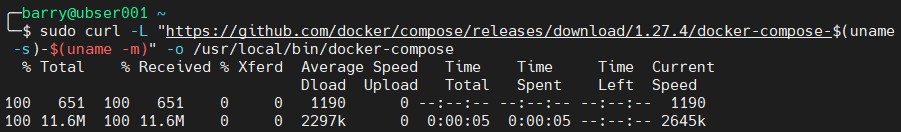
Docker Compose allows users to take multiple containers and integrate them into a single application. Docker Compose uses the YAML format to create the compose files that can be easily executed using docker-compose up or down commands that will create or remove all the containers and configurations within a compose file, respectively.

Let’s install Docker Compose on the Ubuntu server.

1. Install the current stable release of Docker Compose.

Copy

sudo curl -L "https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose



1. Apply executable permissions for the downloaded binary.

Copy

>sudo chmod +x /usr/local/bin/docker-compose

https://s7280.pcdn.co/wp-content/uploads/2020/12/key-87.png

1. Verify the Docker Compose installation.

Copy

docker-compose --version

https://s7280.pcdn.co/wp-content/uploads/2020/12/key-88.png

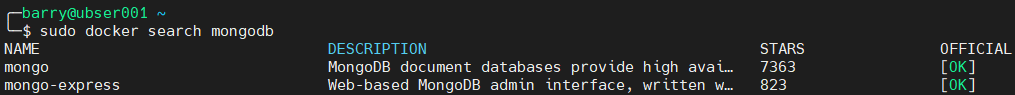
Setting up a MongoDB container

This section will cover how to set up a MongoDB container using a Docker Compose file.

Before creating the compose file, let’s search for the official MongoDB container image using the **search** command.

Copy

sudo docker search mongodb



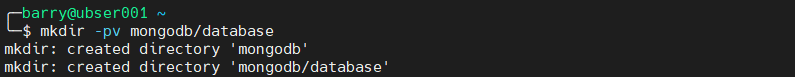
The search results show us that an official MongoDB container image called mongo exists in the docker container registry.

By default, the MongoDB container stores the databases within the /data/db directory within the container.

Next, we need to create a directory called “mongodb” to hold the docker-compose file. We will create another directory called “database” inside the “mongodb” directory to map to the database location of the container. This will enable local access to the database. We use the **-pv** operator to create those parent folders.

Copy

>mkdir -pv mongodb/database



The following docker-compose.yml file will be created within the “mongodb” directory to construct the MongoDB container.

docker-compose.yml

Copy

version: "3.8"

services:

mongodb:

image : mongo

container\_name: mongodb

environment:

- PUID=1000

- PGID=1000

volumes:

- /home/barry/mongodb/database:/data/db

ports:

- 27017:27017

restart: unless-stopped

We used version 3.8 to create the above compose file. The compose file version directly correlates to:

* Which options are available within the compose file
* The minimum supported Docker engine version

In this case, It’s Docker Engine 19.03.0 or newer.

In the compose file, we have created a service called mongodb using the Docker image mongo. We have named the container “mongodb” and mapped the database folder within the container to the local database folder (/home/barry/mongodb/database). These kinds of mappings are known as **bind-mount volumes**.

The environment variables are used to define the “user” and “group” of the container. Finally, we mapped the local port 27017 to internal port 27017. Then the restart policy is set to restart unless stopped by the user.

Here’s the file structure of the project:

Copy

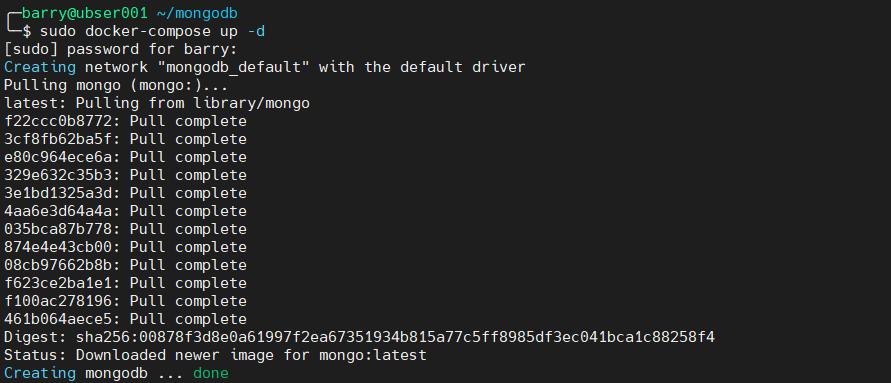
tree mongodb



Go to the “mongodb” folder and run the **docker-compose up** command to start the MongoDB container. The **-d** operator runs the detached container as a background process.

Copy

sudo docker-compose up -d

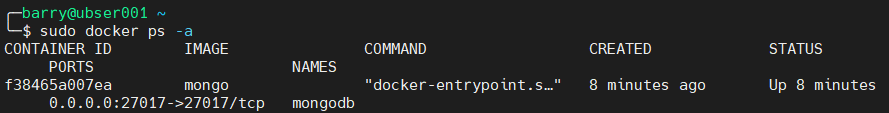


The **up** command will pull the mongo image from the docker registry and create the container using the given parameters in the docker-compose.yml file.

Let’s verify if the container is running and the local folder is populated with the following commands. The **-a** operator will display all the containers within the system regardless of their status.

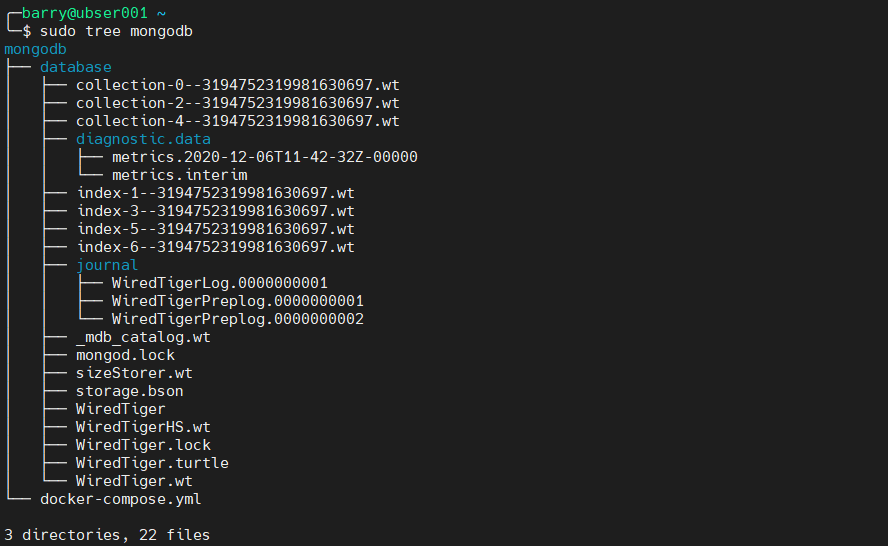
Copy

sudo docker ps -a



Copy

sudo tree mongodb

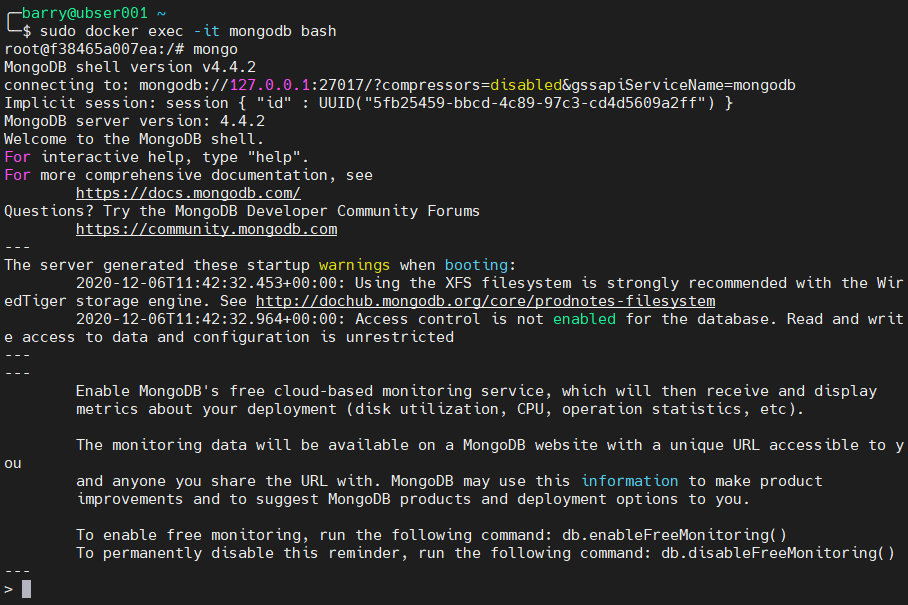


Interacting with the MongoDB container

Using the docker **exec** command, we can access the terminal of the MongoDB container. As the container runs in a detached mode, we will use the Docker interactive terminal to establish the connection.

Copy

sudo docker exec -it mongodb bash



In the bash terminal of the container, we call the mongo command to access MongoDB. We will create a database called “food” and a collection called “fruits”, along with three documents.

1. Switch the database.

Copy

use food

1. Create collection.

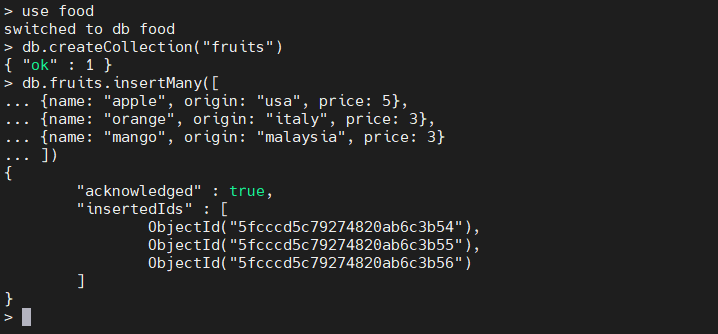
Copy

db.createCollection("fruits")

1. Insert documents

Copy

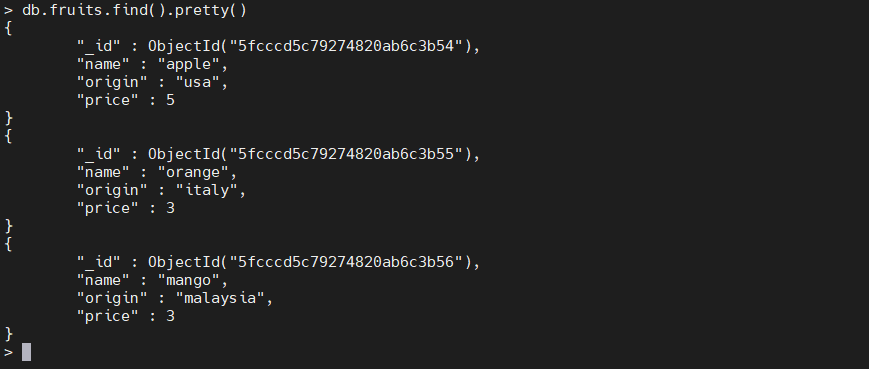
db.fruits.insertMany([ {name: "apple", origin: "usa", price: 5}, {name: "orange", origin: "italy", price: 3}, {name: "mango", origin: "malaysia", price: 3} ])



Search for the documents using the **find** command:

Copy

db.fruits.find().pretty()



The MongoDB container will act like any normal MongoDB installation without any concerns about the underlying software and hardware configuration. Using the **exit** command, we can exit both the MongoDB shell and container shell.



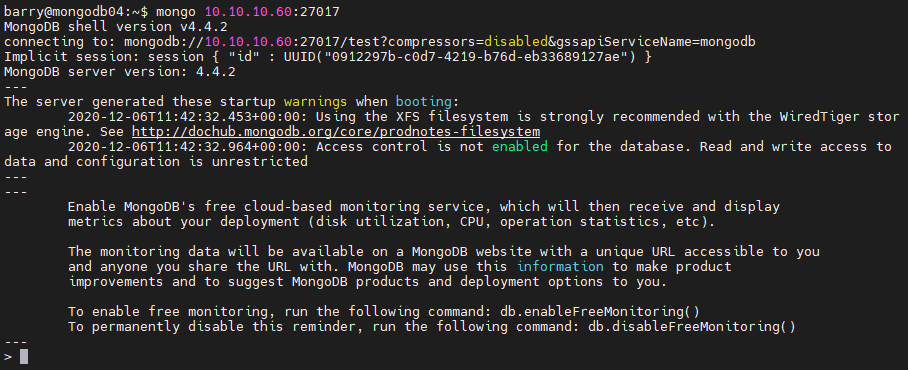
External connections to MongoDB container

While creating the MongoDB container, we mapped the internal MongoDB port to the corresponding port in the server, exposing the MongoDB container to external networks.

The following example demonstrates how we can connect to the container from an external endpoint by simply pointing the **mongo** command to the appropriate server and port.

Copy

mongo 10.10.10.60:27017



The **find** command will search for the fruits collection and its documents to verify that we are connected to the MongoDB container.

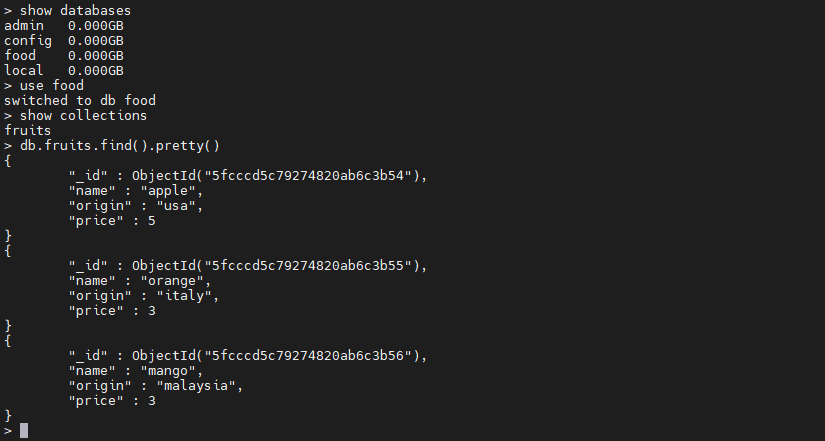
Copy

show databases

use food

show collections

db.fruits.find().pretty()

  
Data resilience

We’ve mapped the database to a local folder. As a result of that, even if the container is removed, the saved data in the local folder can be used to recreate a new MongoDB container.

Let’s test that. We’ll:

* Remove the container using the **docker-compose down**
* Delete the associated images.
* Recreate a new MongoDB database using the compose file and local database files.

Remove the MongoDB container.

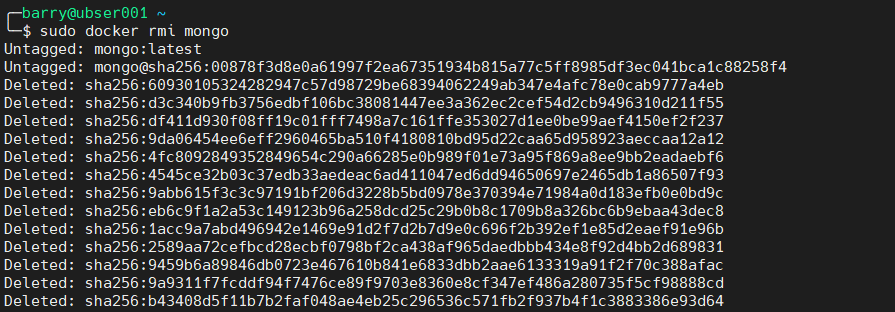
Copy

sudo docker-compose down

  
Remove the local mongo image.

Copy

sudo docker rmi mongo

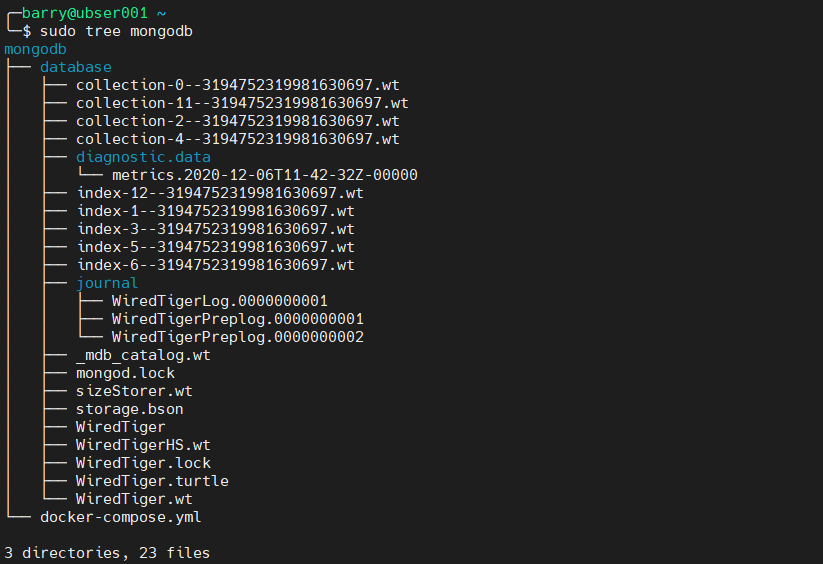


Verify the local database files.

From the output below, we can identify that even though we removed the containers, the data mapped to a local directory did not get removed.

Copy

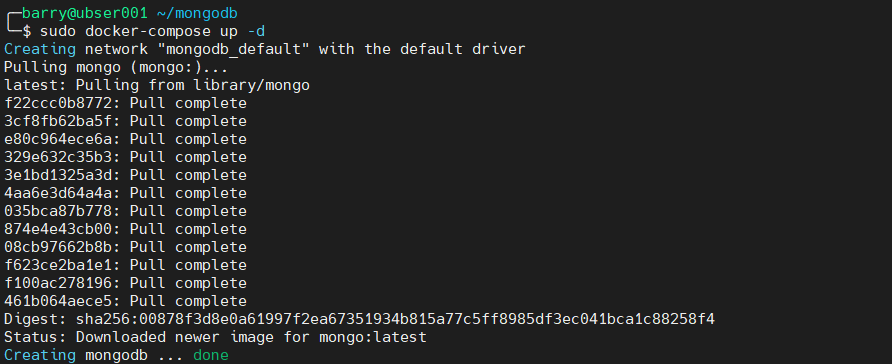
sudo tree mongodb



Recreate a new MongoDB container. Now, we will recreate the container using the original docker-compose.yml file. We execute the following command in the mongodb folder.

Copy

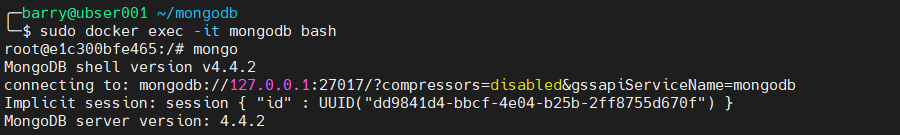
sudo docker-compose up -d



Verify the Data in the MongoDB container. Let’s now access the bash shell in the container and check for the “fruits” collections.

Copy

sudo docker exec -it mongodb bash

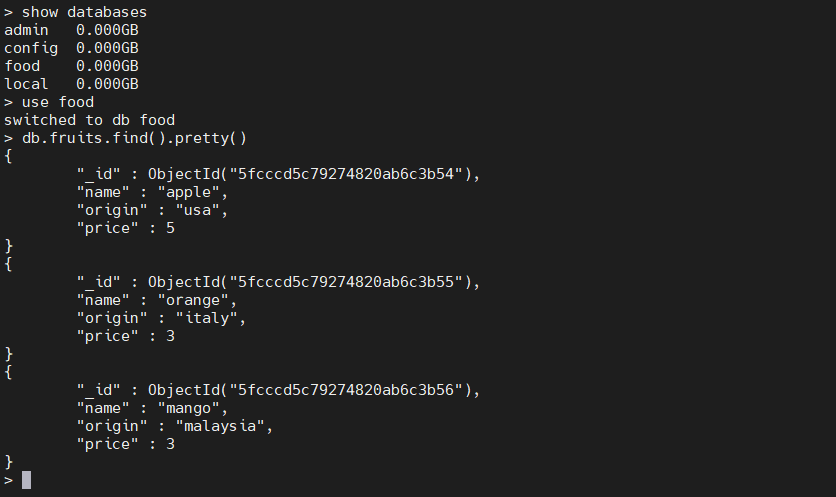


Copy

show databases

use food

db.fruits.find().pretty()



The result indicates that the new container was created with the local database information associated with the new container.

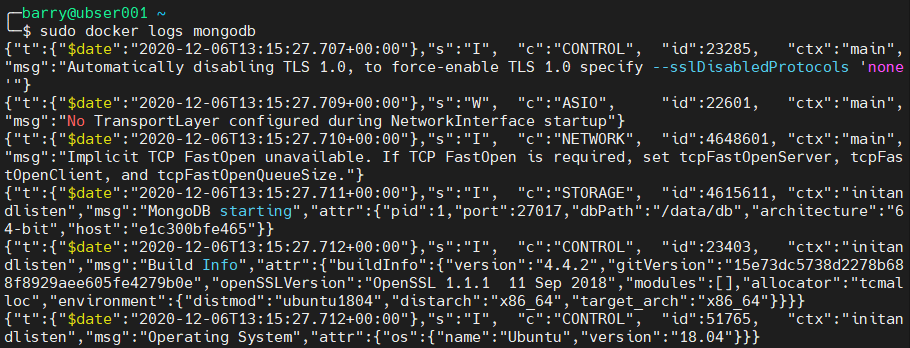
Additionally, we can simply move the container by moving the local folder structure to a new server and creating a container using the docker-compose.yml file. Docker volumes can be used instead of locally saving the data to increase the portability of the database.

Container log files

Every container creates logs that can be used to monitor and debug itself. We can access the container logs using the **docker logs** command with the container name to be monitored.

Copy

sudo docker logs mongodb



Advanced container usage

In this section, we will create a secure MongoDB container that requires a username and password to access the database.

In earlier examples, we mapped the database data to a local folder. However, this is tedious and requires manual intervention when moving the Docker container. Using Docker volumes, we can create Docker native persistent volumes that can be easily transferred between Docker installations.

Although we can use the CLI to manipulate the MongoDB instance, a GUI would be a more convenient option to do that. Mongo Express is a web-based MongoDB administration interface that also can be run as a containerized application.

The docker-compose file comes in handy as a single YAML file that captures all the requirements.

docker-compose.yml

Copy

version: "3.8"

services:

mongodb:

image: mongo

container\_name: mongodb

environment:

- MONGO\_INITDB\_ROOT\_USERNAME=root

- MONGO\_INITDB\_ROOT\_PASSWORD=pass12345

volumes:

- mongodb-data:/data/db

networks:

- mongodb\_network

ports:

- 27017:27017

healthcheck:

test: echo 'db.runCommand("ping").ok' | mongo 10.10.10.60:27017/test --quiet

interval: 30s

timeout: 10s

retries: 3

restart: unless-stopped

mongo-express:

image: mongo-express

container\_name: mongo-express

environment:

- ME\_CONFIG\_MONGODB\_SERVER=mongodb

- ME\_CONFIG\_MONGODB\_ENABLE\_ADMIN=true

- ME\_CONFIG\_MONGODB\_ADMINUSERNAME=root

- ME\_CONFIG\_MONGODB\_ADMINPASSWORD=pass12345

- ME\_CONFIG\_BASICAUTH\_USERNAME=admin

- ME\_CONFIG\_BASICAUTH\_PASSWORD=admin123

volumes:

- mongodb-data

depends\_on:

- mongodb

networks:

- mongodb\_network

ports:

- 8081:8081

healthcheck:

test:  wget --quiet --tries=3 --spider http://admin:admin123@10.10.10.60:8081 || exit 1

interval: 30s

timeout: 10s

retries: 3

restart: unless-stopped

volumes:

mongodb-data:

name: mongodb-data

networks:

mongodb\_network:

name: mongodb\_network

Now, let’s break down the compose file given above. First, we have created two services:

* mongodb
* mongo-express

mongodb service

The root username and password of the mongodb container are configured using the following environment variables.

* MONGO\_INITDB\_ROOT\_USERNAME
* MONGO\_INITDB\_ROOT\_PASSWORD

The data volume is mapped to mongodb-data docker volume, and the network is defined as mongodb\_network while opening port 27017.

mongo-express service

The environment variables of the mongo-express container are:

* ME\_CONFIG\_MONGODB\_SERVER – MongoDB service (mongodb)
* ME\_CONFIG\_MONGODB\_ENABLE\_ADMIN – Enable access to all databases as admin
* ME\_CONFIG\_MONGODB\_ADMINUSERNAME – Admin username of the MongoDB database
* ME\_CONFIG\_MONGODB\_ADMINPASSWORD – Admin password of the MongoDB database
* ME\_CONFIG\_BASICAUTH\_USERNAME – Mongo-Express web interface access username
* ME\_CONFIG\_BASICAUTH\_PASSWORD – Mongo-Express web interface access password

Additionally, we have configured the mongo-express service to depend on the mongodb service. The network is assigned the same mongodb\_network, and the volumes are mapped to mongodb-data volume. Then the port 8081 is exposed to allow access to the web interface.

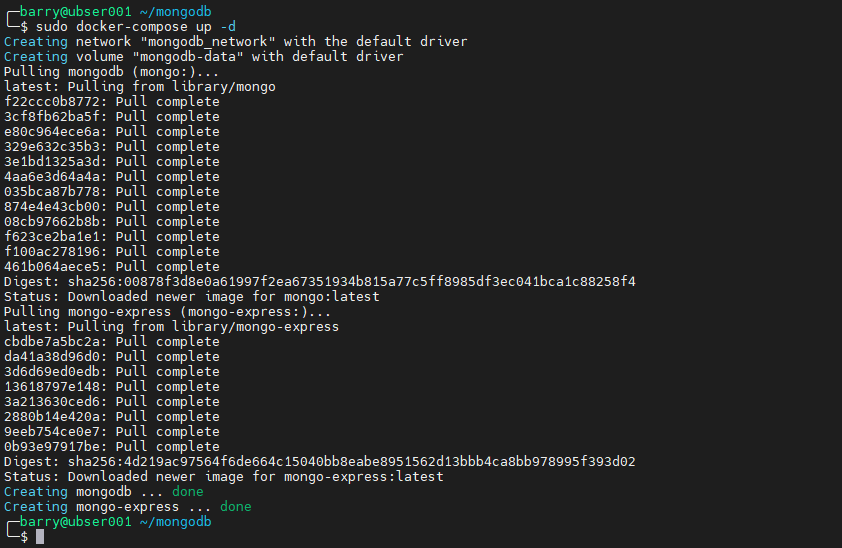
Both services are monitored using Docker health checks. The mongodb service will ping the MongoDB database, while the mongo-express service will try to access the web page using the given credentials.

Finally, we have defined a volume called mongodb-data and a network called mongodb\_network for the project.

Start the Docker compose file.

Copy

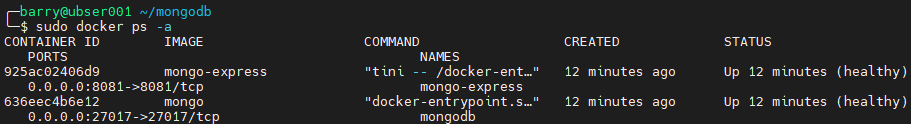
sudo docker-compose up -d



The above output contains no errors. So, we can assume that all the services are created successfully. As we have added health checks for both services, we can verify it by using the docker **ps** command.

Copy

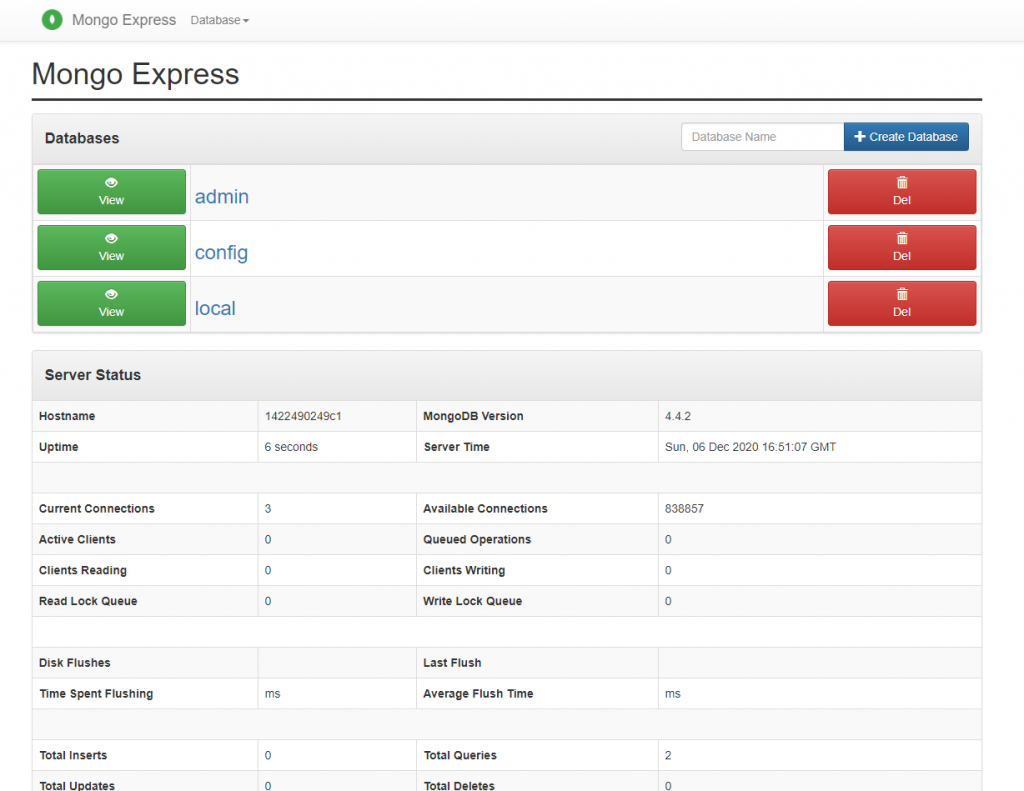
sudo docker ps -a



The docker **ps** command prints the health status of the container. This health status is only available if you have defined a health check for the container.

Mongo Express

Now, let’s go to the Mongo Express web interface using the server IP (http://10.10.10.60:8081).



The Mongo Express interface provides a convenient way to interact with the MongoDB database. The Mongo Express interface also provides an overview status of the MongoDB server instance, providing a simple monitoring functionality.

That concludes this tutorial.