Docker Workshop
Lab 5
Working with Docker networking

1 References

The official reference for Docker networking:

A useful beginner's overview tutorial.

2 Commands covered

3 Lab setup

This is identical to the previous lab

The root folders for all the various Node projects here can be found in the `Lab 5` subfolder in the main `labcode` folder of your downloaded zip for this workshop.

4 Docker Networking basics

Docker comes with 5 built-in network drivers, each serving different purposes.

a) The `bridge` driver (the default) creates a private, internal network on the host. Containers on the same bridge network can communicate via container names, rather than using the localhost. This is the most common use case situation for Docker networking, where multiple containers need to communicate with each other on the same Docker platform.
b) The `host` driver removes network isolation between the container and the host. The container shares the host's networking namespace. This is typically used for high performance networking where container needs full access to the host's network
c) `None` disables networking for a container so it cannot communicate with any other container. This is for containers running isolated workloads that should not communicate for security or efficiency reasons.

The last 2 drivers: `overlay` and `macvlan` are used for complex use cases that are rarely found in practice.

The easiest and most straightforward way for apps in 2 or more containers to communicate with each other is to

a) Create a custom bridge network
b) Run all the apps in containers connected to the same network and provide them with specific user-defined container names
c) In any application code or configuration setting where we need to reference an app for communication purposes (e.g. making HTTP calls), we use the container name for that app.

We will see this in an upcoming lab session.

5 Communication with the external web (HTTP API calls)

The root folder for this project is: `networks-demo`
Open this folder in VS Code.

The implementation of the web app here `app.js` will make REST API calls to a public dummy API server.
https://swapi.dev/

You can test out making HTTP GET requests to these supported API endpoints using a browser. The result is returned as standard JSON content:

https://swapi.dev/api/people

https://swapi.dev/api/people/1

https://swapi.dev/api/people/2

https://swapi.dev/api/films

https://swapi.dev/api/films/1

Whenever this webapp receives GET requests to its 2 internal API endpoints (`/people` and `/films`), it will make a HTTP GET request to those 2 corresponding API endpoints on the dummy server and return the response it gets as its own response (in other words, acting as a simple proxy of sorts for the API endpoints of the actual dummy server).

In addition, it also receives POST requests with JSON in the body and will attempt to connect to a local MongoDB database service to store the JSON content into the service. It will check whether the JSON content is formatted correctly according to an expected model and whether that model already exists in the MongoDB database before storing.

In the project root folder `networks-demo`, create a custom image by building the existing Dockerfile

```
docker build -t favorites-node  .
```

Create a container in detached mode from this custom image and check that it is up:

```
docker run --name favorites -d -p 3000:3000 favorites-node

docker ps
```

Open the URLs below in a browser tab, which will then send HTTP request to the app in container at these two API endpoints, which in turn map to actual HTTP REST API calls to the dummy API server, as explained previously:

http://localhost:3000/movies

http://localhost:3000/people

This demonstrates containers running on a host machine with an Internet connection **can directly communicate with the Web out of the box** without any further additional configuration or change to the existing source code.

```
……
const basicHttps = axios.create({
  httpsAgent: new https.Agent({ rejectUnauthorized: false  })
});
…..

    const response = await basicHttps.get('https://swapi.dev/api/films');

……

    const response = await basicHttps.get('https://swapi.dev/api/people');
```

6 Container to container communication

A common use case is for containers running on the same host machine to communicate with each other. For e.g. many 3 tier web applications will have the backend service that executes the core business logic running in a container, and persistent data for the application will be stored in a standalone SQL or NoSQL database application that will run in a separate container. These two containers will thus be frequently interacting with each other through the life cycle of the app execution

6.1 Connecting via container IP address

Start up a MongoDB container using the official DockerHub image
https://hub.docker.com/_/mongo

```
docker run -d --name mongodb mongo
```

Inspect the container we just started:

```
docker inspect mongodb
```

Go to Network Settings, and determine the IPAddress value, which is the IP address of the container which can be used to reach it. If this is not clear, you can use a Go template expression to drill down and locate this value.

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' mongodb
```

Add this following piece of code in the appropriate location in `app.js` to establish a connection to the MongoDB database service. Include the IP address that you obtained earlier.

```
....

....
// Making the connection to the MongoDB application
mongoose.connect(
  'mongodb://IP-address-for-MongoDB-container:27017/swfavorites',
  { useNewUrlParser: true,  useUnifiedTopology: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      console.log("Connected successfully to MongoDB container !")
    }
  }
);

app.listen(3000);
```

Remove the Node app backend container (it should also be automatically removed):

```
docker rm -f favorites
```

Rebuild the image to incorporate the latest source code changes:

```
docker build -t favorites-node  .
```

The recreate the container based on this new image and check the logs that verify console output that establishes a successful connection was made to the MongoDB container:

```
docker run --name favorites -d -p 3000:3000 favorites-node
```

```
docker logs favorites
```

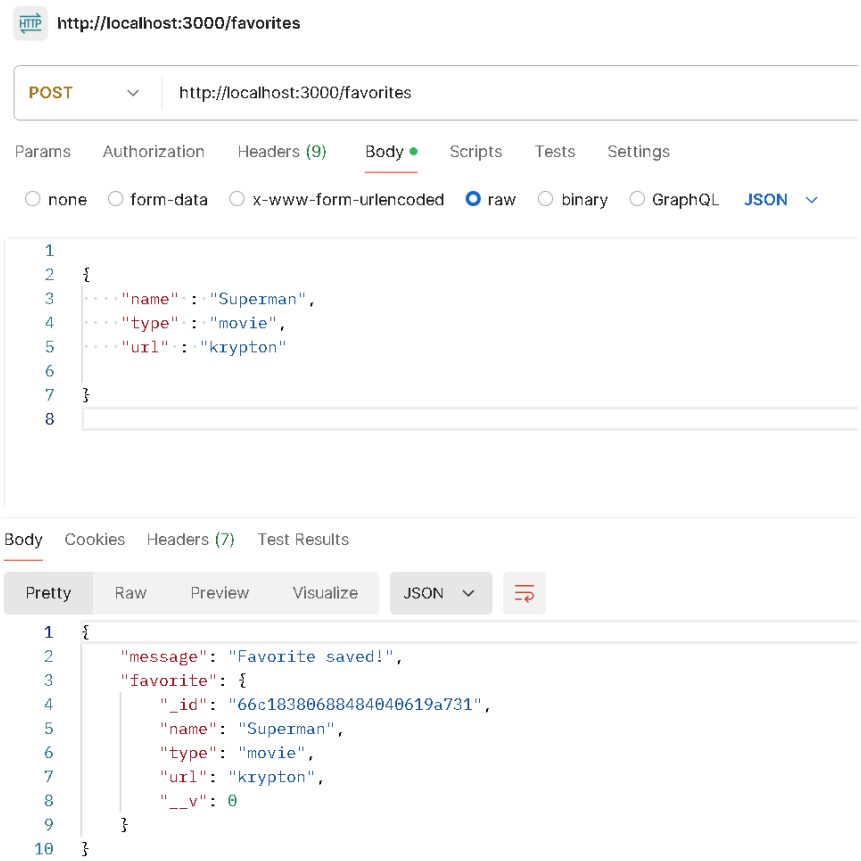
With the MongoDB container still running, we can now make requests to API endpoints for which the app implementation connects to the MongoDB container that is running.

http://localhost:3000/favorites

The response for the first time request will be initially an empty array, as the MongoDB database is initially empty.

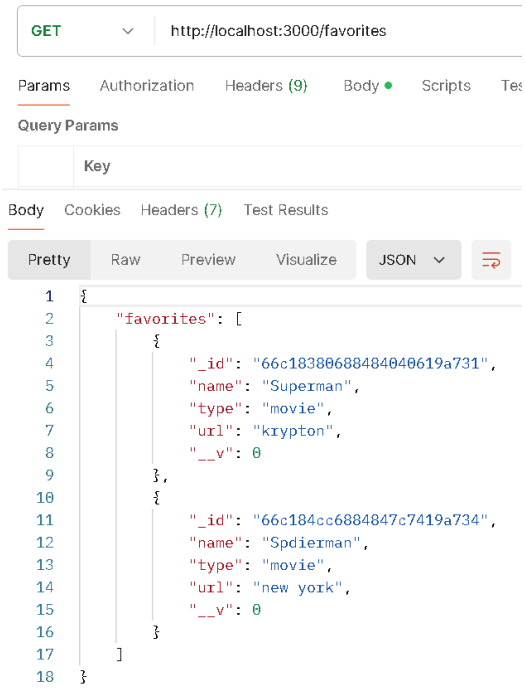
```
{"favorites":[]}
```

We can use a REST client (such as POSTMAN) to send JSON content in the body of a POST request to the same API endpoint to populate the database with some random content. The JSON content should contain the fields: `name`, `type` and `url` with any associated values you wish. If the content is the correct format, you should receive back a response indicating that it has been successfully saved.



You can continue to send a few more POST requests with dummy content to populate the MongoDB database with random content.

Finally, send a GET request to the same endpoint to retrieve all the content stored so far.



Hardcoding the IP address directly into the source code of `app.js` is not the best way to go, since it is possible for the IP address of the MongoDB container to change when it is removed and started up again. This means we have to change the IP address in the source code of `app.js` and rebuild the image again.

6.2 Connecting via container names in a Docker network

Stop the 2 currently running containers and remove them:

```
docker rm -f mongodb
```

```
docker rm -f favorites
```

Now create a network explicitly.

```
docker network create favorites-net
```

Check all existing networks to see that this new network has been created successfully:

```
docker network ls
```

If you have the Docker extension for VS Code installed, you should be able to see this listed in the Network section.

Now start the MongoDB container and place it within this newly created network:

```
docker run -d --name mongodb --network favorites-net mongo
```

If you now inspect the network:

```
docker network inspect favorites-net
```

You will see that the container `mongodb` is associated with it.
Other relevant information includes the subnet range for this network, the IP address for its gateway (IPAM.Config.Subnet / Gateway) as well as the IP address for the `mongodb` container within this subnet.

If you have the Docker extension for VS Code installed, you should be able to inspect the network contents and have this listed in a JSON file.

You can also inspect the network settings of this container with:

```
docker inspect mongodb
```

Go to Network Settings, and check that the nested child Networks value is actually `favorites-net`. If this is not clear, you can use a Go template expression to drill down and locate this value.

```
docker inspect --format '{{ .NetworkSettings.Networks }}' mongodb
```

If two containers are part of the same network, you can **use each container's name directly in the host portion of the URL to address each other directly**. This container name will automatically be translated by Docker into the IP address of that container.

Now change `app.js` to reflect this:

```
// Making the connection to the MongoDB application

mongoose.connect(
  'mongodb://mongodb:27017/swfavorites',
  { useNewUrlParser: true,  useUnifiedTopology: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      console.log("Connected successfully to MongoDB container !")
    }
  }
);
```

Rebuild the image again:

```
docker build -t favorites-node  .
```

and start the Node application again and place it into the same network and verify from the console output that it connected again successfully to the MongoDB container:

```
docker run --name favorites -d -p 3000:3000 --network favorites-net
favorites-node

docker logs favorites
```

You can repeat the previous example of storing dummy data by sending POST requests using your REST client to this API endpoint:

http://localhost:3000/favorites

and then subsequently retrieving all the data using a GET request to the same API endpoint.

The other thing to note is that we did not need to publish any port (using the -p option) for the MongoDB container when we start it, because this container is never ever directly connected to via the `localhost` machine (for which a port mapping is required), instead it is only contacted by the `favorites` container which is running in the same internal network as itself.

Finally stop and remove both containers:

```
docker rm -f mongodb
```

```
docker rm -f favorites
```

7 3-tier web app implemented as multi container application

7.1 Configuring containers to communicate via localhost

The top-level folder for this project is: `multi-demo` which contains the root folders for the backend app project (`/backend`) and front end app project (`/frontend`)

Open this top-level folder `multi-demo` in VS Code.

Start a MongoDB container and map the internal port it is listening on to a `localhost` port of the native host system. We are doing this so that the backend container that we will start later can communicate with it directly on `localhost` as well:

```
docker run --name mongodb --rm -d -p 27017:27017 mongo
```

You can test out that the `MongoDB` container is actively listening on its default native port 27017 by accessing this URL:

```
http://localhost:27017/
```

This just returns a message similar to the one below:

```
It looks like you are trying to access MongoDB over HTTP on the native
driver port.
```

Next, we create an image that we will use to generate the backend container using the existing Dockerfile in /`backend`

Open a Powershell terminal in this folder (/`backend`), and build a custom image from it:

```
docker build -t goals-node .
```

Then attempt to run a container from this image with:

```
docker run --name goals-backend --rm goals-node
```

After a couple of seconds, this will crash and emit some error messages to the console because it fails to connect to MongoDB, and this is because this Node application is now running within a container and is isolated from the other MongoDB container which is only available on the networking system of the native host.

Although the URL for the connection to MongoDB in `app.js` uses localhost, this actually **references the internal networking system of the container, rather than the native host**.

….

```
mongoose.connect(
  'mongodb://localhost:27017/course-goals',
  {

….
```

We can instead substitute a **special domain name** that allows the app within a container to access all ports available on our native host machine running the Docker engine:

```
mongoose.connect(
  'mongodb://host.docker.internal:27017/course-goals',
```

Repeat the build again:

```
docker build -t goals-node .
```

Then run a container from it again but this time in detached mode, and check the console output to verify that it connects successfully to the MongoDB container:

```
docker run -d --name goals-backend goals-node

docker logs goals-backend
```

If you are running Docker on a Linux native host, you will additionally need to add the following flag to enable this special domain name translation to work:

```
docker run -d --name goals-backend --add-host=host.docker.internal:host-gateway goals-node
```

To prepare the backend `goals-backend` container for connection from the front-end container that we will start next, we will first remove it and restart it again with a mapping of the internal port it is currently listening on (80) to an external port on the native localhost.

```
docker rm -f goals-backend

docker run -d --name goals-backend --rm -p 5000:80 goals-node
```

Finally, we create an image that we will use to generate the frontend container. The root project folder for this is `/frontend`

Open a separate and different Powershell terminal in this root project folder (`/frontend`), and use the existing Dockerfile here to build the custom image:

```
docker build -t goals-react .
```

Run a container from the image with:

```
docker run --name goals-frontend --rm -p 3000:3000 goals-react
```

You will get some warning messages regarding deprecated messages which comes from the installation of certain outdated packages. You can ignore this.

After a while, you should see some messages from the development server about serving up the app at localhost:3000

You should be able to access the front end app from the React development server running with the container at: http://localhost:3000 since you have already mapped the internal port 3000 to the same port number on your local host.

If the React server does not serve up the front end app properly to the browser, you may need to rerun it with the additional `-it` option to make it run in the interactive mode:

```
docker run --name goals-frontend --rm -p 3000:3000 -it goals-react
```

When you can see the app served up properly in your browser, you can test out its functionality.

You should be able to add goal items which subsequently appear in the list below, and click on any of the items to remove them.

If you check the existing code in `frontend/src/App.js`, it uses the address `localhost` when attempting to connect to the backend container

```
const response = await fetch('http://localhost:5000/goals', {
```

This works fine since we started the backend container with a mapping of internal port 80 to port 5000 on `localhost`.

At this point, all 3 containers should be able to communicate to each through the localhost network and the bridge network Docker driver. The MongoDB container is running on port 27017 on `localhost`, the backend container is running on port 5000 on `localhost`, while the front end server is serving up the React app to the browser from port 3000 on localhost.

You can verify this by checking for processes and ports on your Windows host.

- For a Docker Desktop installation, Docker containers are associated with the `com.docker.backend.exe` process.
- For a Rancher Desktop installation, Docker containers are associated with the `host-switch.exe` process.

Open a command prompt with admin privilege, and check for the PID of all processes with the appropriate name:

```
tasklist | findstr "process name for rancher or docker"
```

Copy down the PID of all these processes. Then locate all the ports that these processes run on with:

```
netstat -abon | findstr PID
```

You should see the 3 ports mentioned above in the listing:

```
  TCP    0.0.0.0:3000           0.0.0.0:0              LISTENING       PID
  TCP    0.0.0.0:5000             0.0.0.0:0              LISTENING       PID
  TCP    0.0.0.0:27017          0.0.0.0:0              LISTENING       PID
```

You may also see some additional ports depending on whether you are using Rancher Desktop or Docker Desktop. For e.g. Rancher Desktop with Kubernetes and the Trafik Ingres Controller enabled will additionally bind to port 80 and 443.

Remove all these 3 running containers:

```
docker rm -f goals-backend

docker rm -f goals-frontend

docker rm -f mongodb
```

7.2 Configuring containers to communicate via a Docker network

Create a new Docker network:

```
docker network create goals-net
```

Verify that it is created:

```
docker network ls
```

Now we run all the containers within the same network. As you can recall, from the previous lab sessions, this allows us to directly use the container name

Start with the MongoDB container within this network.

```
docker run --name mongodb --rm -d --network goals-net mongo
```

Notice that we did not publish the internal port of 27017 that MongoDB is listening on to a localhost port because we are now expecting the backend container to communicate to this MongoDB container using its container name directly within this new network.

Moving on to the backend Node application, we will need to modify `backend/app.js` here so that it connects to the MongoDB container using the assigned name (`mongodb`):

```
mongoose.connect(
  'mongodb://mongodb:27017/course-goals',
```

Rebuild the image again:

```
docker build -t goals-node .
```

Finally start the container from this new image within the new network that we just created:

```
docker run --name goals-backend --rm -d --network goals-net goals-node
```

Verify that it connected successfully to the MongoDB container using the assigned name (`mongodb`) that we modified in our source code:

```
docker logs goals-backend
```

As we are intending to place our front-end container within the same network as the back-end container and the MongoDB container, we will naturally expect the same technique to work: which is we can now replace all references to `localhost` with the name of the backend container (`goals-backend`)

Perform this replacement now in `frontend/src/App.js`

```
.....
  useEffect(function () {
    async function fetchData() {
      setIsLoading(true);

      try {
        const response = await fetch('http://goals-backend:5000/goals');
......

  async function addGoalHandler(goalText) {
    setIsLoading(true);

    try {
      const response = await fetch('http://goals-backend:5000/goals', {

.......


    try {
      const response = await fetch('http://goals-backend:5000/goals/' +
goalId, {
        method: 'DELETE',
      });


.......
```

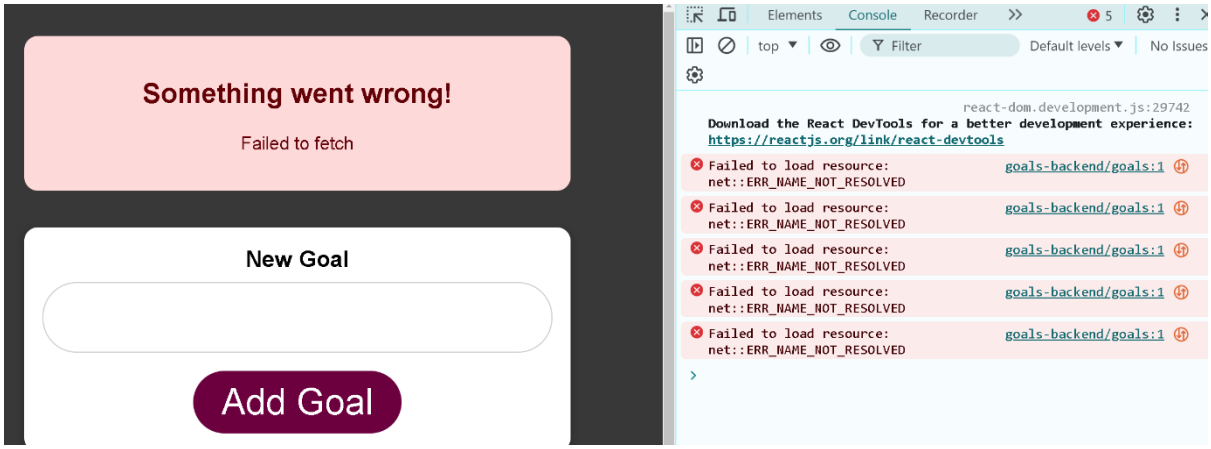
Then rebuild the image:

```
docker build -t goals-react .
```

When we create a new container from this image, we will still retain the previous port mapping in order to allow us to interact from our browser on localhost with the React development server in the container at port 3000, but we will also want to ensure that this container remains in the same network so that our new code change will work:

```
docker run --name goals-frontend --rm -p 3000:3000 --network goals-net goals-react
```

However, now unexpectedly in the front end in the browser, we obtain a new error: ERR_NAME_NOT_RESOLVED



This is due to the React code for the front end (`app.js`) is currently executed within the browser environment: it is served up to the browser from the React development server which is running within the container. Any network request made in the React code (such as http://goals-backend/goals/), will have its DNS performed by the host system (Windows or Linux).

In this case, the host system is not able to resolve this to the correct container IP address: only the Docker system will be able to do so. However, Docker is not able to do this now as the `app.js` is **NOT RUNNING** inside a Docker container: it is **running inside a browser which utilizes the native host networking system**.

The only domain name that can be resolved by the native host DNS is `localhost`, so the React front-end code running in the browser must use this. However, the back-end container is currently running within the internal Docker network, so using `localhost` in the front-end code will not allow it to access the backend.

To resolve the dilemma described above, we now need to

- make the backend container accessible on `localhost` through a published port
- revert the code for our front end back to access `localhost` again

We first start off by removing the backend container:

```
docker rm -f goals-backend
```

Now restart it again, but this time with its internal port of 80 additionally published to `localhost` on port 5000 to allow the frontend to access it, but at the same time keeping it within the internal network we had created earlier to allow it to communicate with the mongoDB container:

```
docker run --name goals-backend --rm -d  -p 5000:80 --network goals-net goals-node
```

Finally, we remove the front end container:

```
docker rm -f goals-frontend
```

Next change everything back again to `localhost` in `frontend/src/App.js`

```
.....
  useEffect(function () {
    async function fetchData() {
      setIsLoading(true);

      try {
        const response = await fetch('http://localhost:5000/goals');
......

  async function addGoalHandler(goalText) {
    setIsLoading(true);

    try {
      const response = await fetch('http://localhost:5000/goals', {

.......

    try {
      const response = await fetch('http://localhost:5000/goals/' + goalId,
{
        method: 'DELETE',
      });

.......
```

Then rebuild the image:

```
docker build -t goals-react .
```

Now we can start it again, but this time we no longer need to place the container within the Docker network. This is because the app is running in the browser and can directly communicate with the backend which has already published a port to `localhost` previously:

```
docker run --name goals-frontend --rm -p 3000:3000 goals-react
```

Now you should be able to interact with the front end app and enter new goals items as well as delete existing ones.

7.3 Configuring volumes and authentication credentials via environment variables

Docker supports runtime environment variables which can also be set as options in the Dockerfile and also within the docker run command. This allows you to customize certain variables which are likely to change between different environments that a Docker container is likely to run in or interact with. Examples might be port numbers, authentication credentials for databases, etc

Environment variables allow the creation of more flexible images and containers because you don't have to hard-code everything into the Dockerfile for an image. Instead, you can set it dynamically when you build an image or even only when you run a container.

Right now, if we remove the MongoDB container

```
docker rm -f mongodb
```

and rerun it again

```
docker run --name mongodb --rm -d --network goals-net mongo
```

all the data stored within it will be lost, as expected since we are starting this container from scratch. Verify this for yourself by refreshing / reloading the React app from the development server at localhost:3000.

Remove the container again:

```
docker rm -f mongodb
```

The Docker Hub documentation for MongoDB shows the exact path within the container where the data for the database is stored:
https://hub.docker.com/_/mongo
Look at topic: Where to Store Data.
The path where data is stored in the container is `/data/db`

All the official DockerHub database images (mySQL, PostgreSQL, etc) will include documentation that shows the exact path within the container where the data for the database is stored.
As an example, if we check the MySQL documentation
https://hub.docker.com/_/mysql
Look at topic: Where to Store Data.
The path where data is stored in the container is `/var/lib/mysql`

When we next create the container again, we specify a named volume to persist the data from this path:

```
docker run --name mongodb -v data:/data/db --rm -d --network goals-
net mongo
```

You can check for the creation of this new volume with:

```
docker volume ls
```

Enter a few new goal items from the React frontend, and stop the container again

```
docker rm -f mongodb
```

Recreate it using the named volume that we specified earlier for the first time:

```
docker run --name mongodb -v data:/data/db --rm -d --network goals-net mongo
```

You should now be able to see that all the goal items were persisted, due to the named volume as we have already seen before in a previous lab.

Authentication information is also provided at the DockerHub documentation page (see Topic Environment Variables), which explains support for MONGO_INITDB_ROOT_USERNAME and MONGO_INITDB_ROOT_PASSWORD for creating a simple user.

Stop the container again:

```
docker rm -f mongodb
```

Recreate it using the named volume and also adding in the environment variables to specify the user name and password for the first initial use of the database

```
docker run --name mongodb -v data:/data/db --rm -d --network goals-net -e MONGO_INITDB_ROOT_USERNAME=mongoadmin -e MONGO_INITDB_ROOT_PASSWORD=secret mongo
```

Now at this point of time, reloading the React app reports failure in fetching the goal items since we did not provide any authentication credentials with our GET request.

This means we no need to ensure that in the `app.js` for our backend, we utilize this username/password combination every time we make a request to the MongoDB container.

Next we remove the backend container:

```
docker rm -f goals-backend
```

Make the appropriate change to `app.js` to include the username/password combination that we specified when we started the MongoDB container:

https://www.mongodb.com/docs/v4.4/mongo/#mongodb-instance-with-authentication

```
mongoose.connect(
  'mongodb://mongoadmin:secret@mongodb:27017/course-goals?authSource=admin',
```

Hard coding the authentication credentials directly into the source code is not a good security practice, and we will later move these credentials to a local environment file which can be referenced through a CLI option.

For now, we will first rebuild the backend image:

```
docker build -t goals-node .
```

Finally, to get everything to work properly again with authentication in place, we will first stop the MongoDB container, delete the named volume that you are backing up your data with, and then restart the MongoDB container again:

```
docker rm -f mongodb

docker volume rm data

docker run --name mongodb -v data:/data/db --rm -d --network goals-net -e MONGO_INITDB_ROOT_USERNAME=mongoadmin -e MONGO_INITDB_ROOT_PASSWORD=secret mongo
```

Then, restart the backend container in the same way.

```
docker run --name goals-backend --rm -d -p 5000:80 --network goals-net goals-node
```

Confirm that everything is working as usual, but with authentication in place now from the backend container to the MongoDB database.

After entering a few items, you can check that the named volume has persisted the data by first removing the MongoDB container:

```
docker rm -f mongodb
```

And restart a new container in the same way:

```
docker run --name mongodb -v data:/data/db --rm -d --network goals-net -e MONGO_INITDB_ROOT_USERNAME=mongoadmin -e MONGO_INITDB_ROOT_PASSWORD=secret mongo
```

And you should be able to retrieve all the previous data items that you had entered in the front end.

We can now move the authentication credentials that we previously hardcoded into `backend/app.js` into a separate environment file that we store locally on the host machine.

Create a new environment file `.env` in the root folder `backend` and place the actual value for the authentication credentials there

```
.env
```

```
MONGODB_USERNAME=mongoadmin
MONGODB_PASSWORD=secret
```

We modify `backend/app.js` according to access this environment variable values:

```
mongoose.connect(
  `mongodb://${process.env.MONGODB_USERNAME}:${process.env.MONGODB_PASSWORD}
@mongodb:27017/course-goals?authSource=admin`,
```

Remove the backend container

```
docker rm -f goals-backend
```

Rebuild the image again to incorporate the changes to `app.js`

```
docker build -t goals-node .
```

Restart the backend container again with a reference to the `.env` file we created earlier:

```
docker run --name goals-backend --rm -d  -p 5000:80 --env-file ./.env
--network goals-net goals-node
```

And verify that it still remains working (you can insert and delete items from the front end)

When you are satisfied with interacting with the app, you can remove all 3 running containers