

Docker Lab 5

Working with Docker networking

1	REFERENCES	1
2	COMMANDS COVERED	1
3	LAB SETUP	1
4	COMMUNICATION WITH THE EXTERNAL WEB (HTTP API CALLS)	1
5	CONTAINER TO CONTAINER COMMUNICATION	3
5.1	CONNECTING VIA CONTAINER IP ADDRESS	3
5.2	CONNECTING VIA CONTAINER NAMES IN A DOCKER NETWORK	6
6	3-TIER WEB APP IMPLEMENTED AS MULTI CONTAINER APPLICATION	8
6.1	CONFIGURING CONTAINERS TO COMMUNICATE VIA LOCALHOST	8
6.2	CONFIGURING CONTAINERS TO COMMUNICATE VIA A DOCKER NETWORK	11
6.3	CONFIGURING VOLUMES AND AUTHENTICATION CREDENTIALS FOR THE MONGODB CONTAINER	14

1 References

The official reference for Docker networking:

<https://docs.docker.com/network/>

Additional tutorials

<https://spacelift.io/blog/docker-networking>

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 Communication with the external web (HTTP API calls)

The root folder for this project is: networks-demo

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');
```

5 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

5.1 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://172.17.0.3:27017/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, and the type field should either contain movie or character. If the content is the correct format, you should receive back a response indicating that it has been successfully saved.

 **http://localhost:3000/favorites**

POST  **http://localhost:3000/favorites**

Params Authorization Headers (9) **Body** ● Scripts Tests Settings



☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** 

```

1
2 {
3   .... "name" : "Superman",
4   .... "type" : "movie",
5   .... "url" : "krypton"
6
7 }
8

```

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON  

```

1 {
2   "message": "Favorite saved!",
3   "favorite": {
4     "_id": "66c18380688484040619a731",
5     "name": "Superman",
6     "type": "movie",
7     "url": "krypton",
8     "__v": 0
9   }
10 }

```

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.

The screenshot shows a web browser's developer tools with the 'Network' tab selected. A GET request to `http://localhost:3000/favorites` is shown. The response is a JSON array of two movie objects. The first object is for 'Superman' and the second is for 'Spdierman'.

```

1  {
2    "favorites": [
3      {
4        "_id": "66c18380688484040619a731",
5        "name": "Superman",
6        "type": "movie",
7        "url": "krypton",
8        "__v": 0
9      },
10     {
11       "_id": "66c184cc6884847c7419a734",
12       "name": "Spdierman",
13       "type": "movie",
14       "url": "new york",
15       "__v": 0
16     }
17   ]
18 }

```

Hardcoding the IP address directly into the source code of `app.js` is not the best way to go, since the IP address of the MongoDB container is likely 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.

5.2 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
```

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

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

Just as before, you can 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
```

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
```

6 3-tier web app implemented as multi container application

6.1 Configuring containers to communicate via localhost

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

Start a MongoDB container and map the internal port it is listening to a localhost port of the same number so that our backend can communicate with it directly on localhost as well:

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

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

In a Powershell terminal in this folder, 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 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 a Docker container to access all ports available on our local 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 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
```

Finally, we create an image that we will use to generate the frontend container:

Use the existing Dockerfile in `frontend` to build the custom image:

In the `frontend` subfolder, generate the image from it with:

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

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

Run a container from the image with:

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

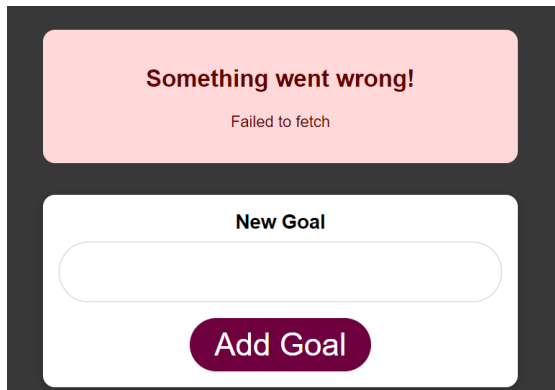
After a while, you should see some messages from the development server about serving up the app at `localhost:3000`

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 the app runs in the browser for the first time, you will notice a fetch error:



This is due to the fact that the existing code in `frontend/src/App.js` uses the address `localhost` when attempting to connect to the backend service

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

However, if you look carefully at the command that we used to previously start the backend container,

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

We **DID NOT** perform any publishing of internal ports of the backend service to `localhost`, and hence the fetch operation to `localhost` (at the default port of 80) will fail.

To resolve this problem, first remove both the front end and back end containers:

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

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

Now restart the backend container with a port mapping to port 80 on the localhost (the default port for HTTP requests):

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

Finally restart the front end container:

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

The app should work fine now. 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.

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 80 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. The Docker containers are run by the `com.docker.backend.exe` process.

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

```
tasklist | findstr "com.docker.backend.exe"
```

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	0.0.0.0:80	0.0.0.0:0	LISTENING	PID
TCP	0.0.0.0:3000	0.0.0.0:0	LISTENING	PID
TCP	0.0.0.0:27017	0.0.0.0:0	LISTENING	PID

Remove all these 3 running containers:

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

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

```
docker rm -f mongodb
```

6.2 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
```

Continue with the backend Node application

We 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
```

As we are intend to place our front-end container within the same network as the back-end container and the MongoDB container, we will repeat the same procedure for the front-end, where we modify `frontend/src/App.js` and replace all `localhost` references with the name of the backend container (`goals-backend`):

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

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

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

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

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

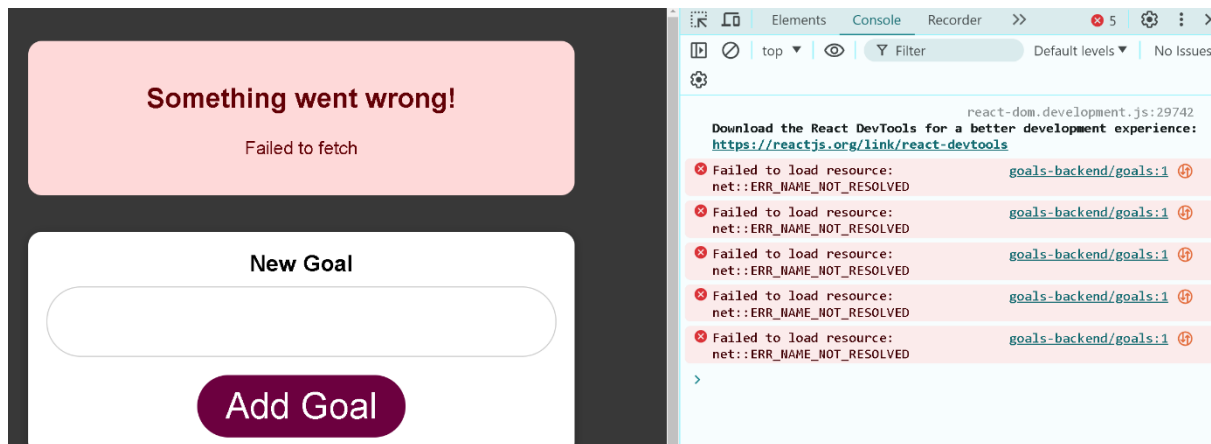
Then rebuild the image:

```
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 ensure that this container remains in the same network:

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

Now in the front end we get a failure to get error due to `ERR_NAME_NOT_RESOLVED`



This is due to the React code for the front end being executed within the browser, and hence when a network request is sent out to <http://goals-backend/goals/>, this will not be resolved to the correct container IP address (which would be the case if the app was running within a container).

The only domain name that can be resolved by the native 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 communicate properly. To resolve this, we now need to also make the backend container accessible on `localhost` through a published port.

We first start off by removing the backend container:

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

Now restart it but make sure its additionally published to `localhost` on port 80 to allow the frontend to access it, but while still 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 80:80 --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/app.js`

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

```

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

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

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

      try {
        const response = await fetch('http://localhost/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 it 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.

6.3 Configuring volumes and authentication credentials for the MongoDB 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.

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
```

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.

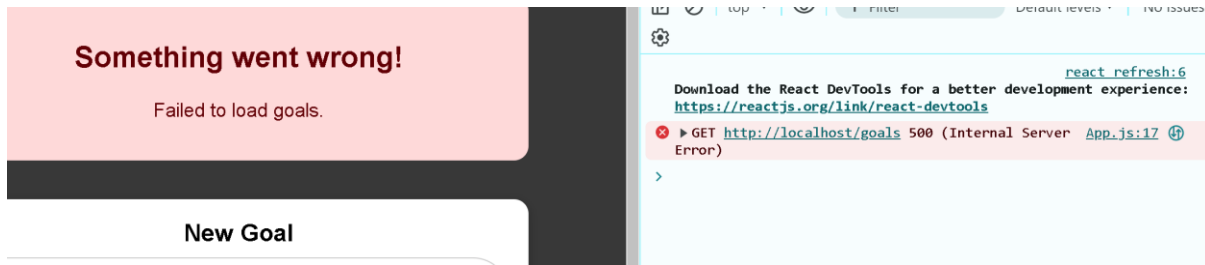
Stop the container again:

```
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, as we have already explained previously, 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 80:80 --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 80:80 --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