Docker Lab 5
Working with Docker networking

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-starting-setup`

This Node web app 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 in the form of JSON.

https://swapi.dev/api/people

https://swapi.dev/api/films
Whenever this webapp receives GET requests to these 2 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-starting-setup`, build the existing Dockerfile

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

Then run a container from the image:

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

The application crashes with an error message referring to the inability to connect to the MongoDB database:

```
MongoNetworkError: failed to connect to server [localhost:27017] on
first connect [Error: connect ECONNREFUSED 127.0.0.1:27017
    at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1605:16)
{
  name: 'MongoNetworkError'
}]
```

This is expected because we don't have a MongoDB database service in operation yet.

Comment out the affected code to verify whether we can run the app and listen at the specified port of 3000:

```
app.listen(3000);
```

```
// mongoose.connect(
//   'mongodb://localhost:27017/swfavorites',
//   { useNewUrlParser: true },
//   (err) => {
//     if (err) {
//       console.log(err);
//     } else {
//       app.listen(3000);
//     }
//   }
// );
```

Rebuild the same image again with:

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

Now attempt to run a container again from this rebuilt image:

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

This time the container stays up and running,

```
docker ps -a
```

Use your browser to send HTTP request to the app in container at these two endpoints, which map to actual HTTP REST API calls to the dummy API server:

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 configuration or settings

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 typical SQL or NoSQL database 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
```

Use this IP address to update the relevant URL in app.js for establishing a connection to the MongoDB database service and uncomment the code so you can apply it.

```
mongoose.connect(
  'mongodb://172.17.0.3:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
```

Stop the container (it should also be automatically removed):

```
docker stop favorites
```

Rebuild the image to incorporate the latest source code changes:

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

The rerun the container based on this new image:

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

This time the container stays up and running, which we can check with:

```
docker ps
```

This indicates the code portion to establish connection to the MongoDB container using the IP address that we acquired from it successfully.
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

This will be initially an empty array, as this MongoDB database is initially empty.

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 will dynamically 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 within the favorites-net network:

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

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:

```
mongoose.connect(
  'mongodb://mongodb:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
```

```
      app.listen(3000);
    }
  }
);
```

Rebuild the image again:

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

and start the Node application again and place it into the same network:

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

Once again, we can now make requests to API endpoints for which the app implementation connects to the MongoDB container that is running in the same internal network as the Node application. For e.g.
http://localhost:3000/favorites

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-01-starting-setup`
which contains the root folders for the backend app project (`/backend`) and front end app project (`/frontend`)

Start a MongoDB container and expose the internal port it is listening to become available on the localhost so that our backend can communicate with it:

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

Next, we create an image that we will use to generate the backend container:

Create a new Dockerfile in `/backend`

Dockerfile

```
FROM node:14

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 80

CMD ["node", "app.js"]
```

We build a new image from it:

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

Then run a container from it 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, but the URL for the connection to MongoDB in `app.js` is still using localhost.

```
….

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:

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

And now you should see it is connected to MongoDB

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

Create a new Dockerfile in `/frontend`

```
FROM node:14

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 3000

CMD [ "npm", "start" ]
```

Build an image from it with:

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

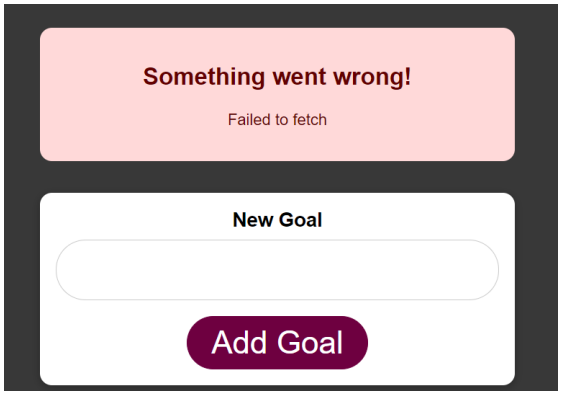
Run a container from the image with:

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

You should be able to access the front end app from the React server 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 --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 connected to the new network that we just created:

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

Check that both the containers are running properly

```
docker ps
```

Finally we 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',
      });


.......



.......
```

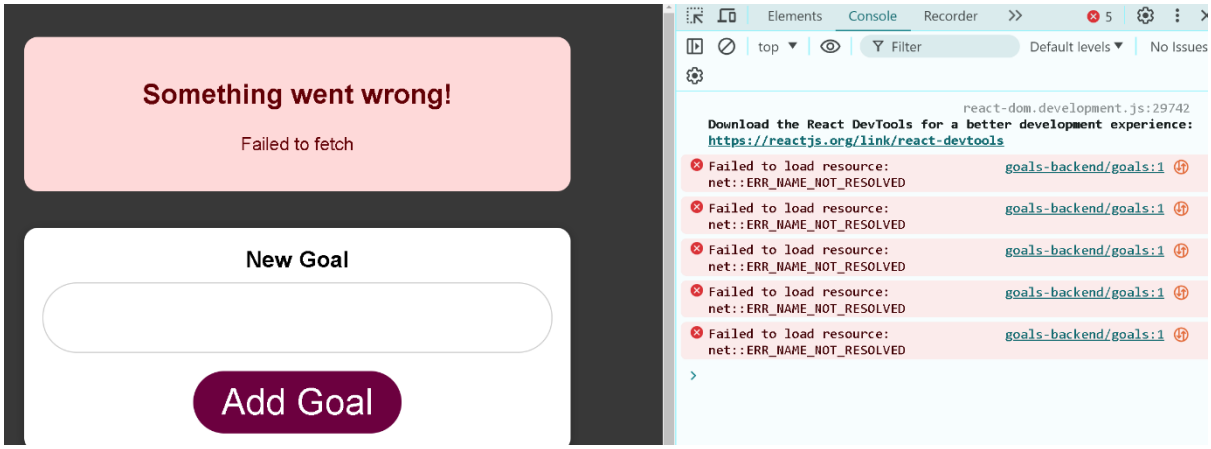
Then rebuild the image:

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

When we run we will still maintain the previous port mapping in order to allow us to interact with the React development server from our localhost, but also to ensure that this container is 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 where a network request is send 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).

So we would need to use localhost instead in the front end code, and somehow make the backend accessible on the localhost through a published port.

First remove the front end container:

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

Next change everything back again to `localhost` in 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 without placing it in the internal network because we can see this is no longer useful or required since the app itself is running in the browser:

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

Next we remove 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
```

Check all containers are running:

```
docker ps
```

And 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 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 is stored:
https://hub.docker.com/_/mongo
Look at topic Where to Store Data.

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

Authentication information is also provided at the DockerHub documentation page, 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 (See the Environment Variables section)

```
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',
```

Rebuild the backend image:

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

and restart it again:

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

This time you should be able to connect from your backend container to MongoDB, and therefore you should be able to add and remove goal items from the front end. If you encounter an authentication error (resulting in the goals-backend container crashing when it starts), you should first stop the MongoDB container, delete the named volume that you are backing up your data with, and then restart the MongoDB container again as shown:

```
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
```

Once you confirm that everything is working, you can later remove 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.

6.4 Persisting data through volumes and bind mounts

Remove the backend container:

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

We will use a named volume here for the backend container to persist the logs that is implemented in appropriate coding in app.js. The `access.log` file is contained in `/logs`

```
……

const morgan = require('morgan');

const accessLogStream = fs.createWriteStream(
  path.join(__dirname, 'logs', 'access.log'),
  { flags: 'a' }
);

app.use(morgan('combined', { stream: accessLogStream }));

……
```

We can now start up the back end container with a mapping for the named volume called `logs`:

```
docker run --name goals-backend -v logs:/app/logs --rm -d  -p 80:80 --network goals-net goals-node
```

Check for the named volume creation with:

```
docker volume ls
```

Connect to the running container and navigate to this folder and inspect the `access.log` file (which is the actual log file that is being dynamically updated in real time) in `/app/logs`

```
docker exec -it goals-backend /bin/bash
```

```
root@bf5ee440e713:/app# cd /app/logs
root@bf5ee440e713:/app/logs# ls -l
total 8
-rwxr-xr-x 1 root root 7945 Aug  6 12:21 access.log
root@bf5ee440e713:/app/logs# cat access.log
```

Keep interacting with the app by adding goals or removing existing ones: remember that each interaction with the front end results in HTTP requests to the back end which are then recorded dynamically in `access.log` (which you can check with repeated **`cat access.log`**).

Exit from the interactive shell after you have interacted for a while, and remove the container:

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

As we have already seen, we can create a light weight container from a basic Linux image like alpine and map that to our named volume and open an interactive shell within it to examine its contents independently from our backend container

```
docker run --rm -it -v logs:/mnt alpine /bin/sh
```

We can further add a bind mount so that every time we update our source code in `backend\app.js`, this is immediately copied over into the corresponding folder in the container. At the same time, we can add an anonymous volume which will reference `node_modules` folder in the container to avoid the non-existing `node_modules` in our root project folder from overwriting the existing one there.

Before running the command below, make sure that the MongoDB container is already up and running.

```
docker run --name goals-backend -v "complete path to root folder:/app"
-v logs:/app/logs -v /app/node_modules --rm -d  -p 80:80 --network
goals-net goals-node
```

or

```
docker run --name goals-backend -v ${pwd}:/app -v logs:/app/logs -v
/app/node_modules --rm -d  -p 80:80 --network goals-net goals-node
```

Verify that it is running and that is also coordinating with the other 2 containers (mongdb and frontend)

Right now, the way we are starting the server app is via node, which does not monitor for changes in real time and so even if we have a bind mount to immediately reflect the latest changes to app.js into

the container file system, node runtime will not immediately pick it up. So we repeat what we did previously to add a dependency for nodemon in package.json in `/backend`

```
.....

  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon -L app.js"
  },
  "author": "Maximilian Schwarzmüller / Academind GmbH",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1",
    "mongoose": "^5.10.3",
    "morgan": "^1.10.0"
  },
  "devDependencies": {
    "nodemon" : "^2.0.4"
  }


.....
```

Also modify the Dockerfile to use `nodemon` to start app.js

```
FROM node:14

WORKDIR /app
```

```
COPY package.json .

RUN npm install

COPY . .

EXPOSE 80

CMD ["npm", "start"]
```

Remove the backend container

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

Rebuild the image again to incorporate the changes to package.json and Dockerfile.

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

And then restart the container with this rebuilt image using the latest options:

```
docker run --name goals-backend -v "complete path to root folder:/app"
-v logs:/app/logs -v /app/node_modules --rm -d  -p 80:80 --network
goals-net goals-node
```

or

```
docker run --name goals-backend -v ${pwd}:/app -v logs:/app/logs -v
/app/node_modules --rm -d  -p 80:80 --network goals-net goals-node
```

Verify that nodemon is actually running to start node.js

```
docker logs goals-backend
```

Make a change to `app.js` (for e.g. a change to the console.log at the end for the mongoose.connect statement) and save

```
  (err) => {
    if (err) {
```

```
      console.error('FAILED TO CONNECT TO MONGODB');
      console.error(err);
    } else {
      console.log('MAKE SOME KIND OF WEIRD CHANGE HERE !!!');
      app.listen(80);
    }
  }
```

Check whether this is immediately reflected in the console output for the backend container

`docker logs goals-backend`

We will replace the hardcoded username and password in `app.js` of the backend with ENV variables since placing username / passwords directly in source code represents a big security vulnerability that is easily exploited.

Start by placing the ENV variables in the Dockerfile of `\backend`

```
FROM node:14
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
EXPOSE 80
ENV MONGODB_USERNAME mongoadmin
ENV MONGODB_PASSWORD secret

CMD ["npm", "start"]
```

We modify app.js according to substitute in this ENV variables:

```
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 and Dockerfile.

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

Restart the backend container again:

```
docker run --name goals-backend -v ${pwd}:/app -v logs:/app/logs -v
/app/node_modules --rm -d  -p 80:80 --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

7 END