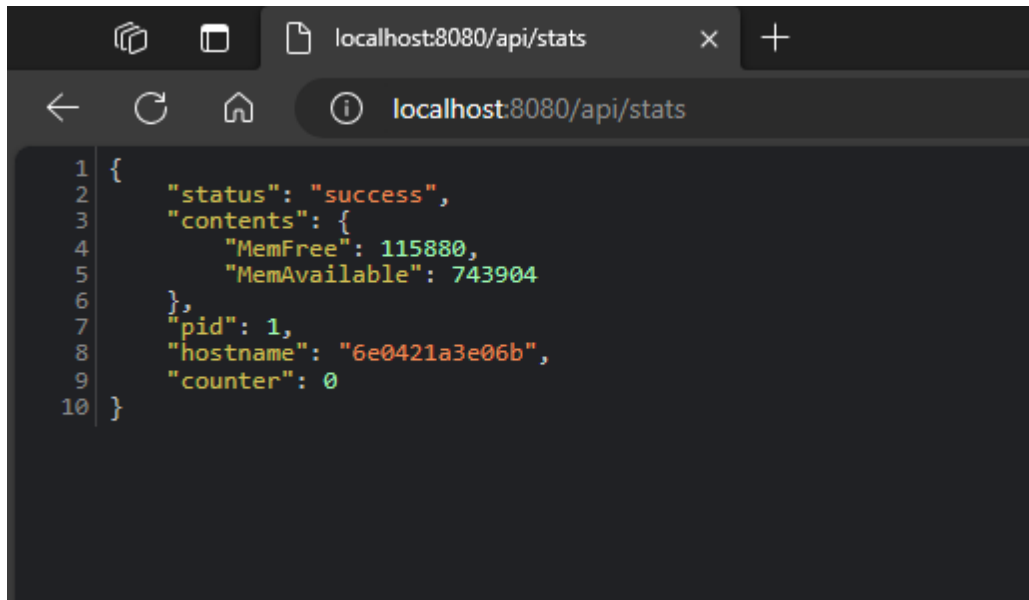


For this journal:

- We will be using the Git Bash terminal. Any other terminal can also be used; e.g., Z Shell (zsh) for Mac users and CMD or Windows PowerShell for Windows users.
- It is assumed you have installed Docker Compose and Docker Desktop on your system. You can follow the guide here ([Overview of installing Docker Compose | Docker Docs](#)) to do so.

Challenge 4:

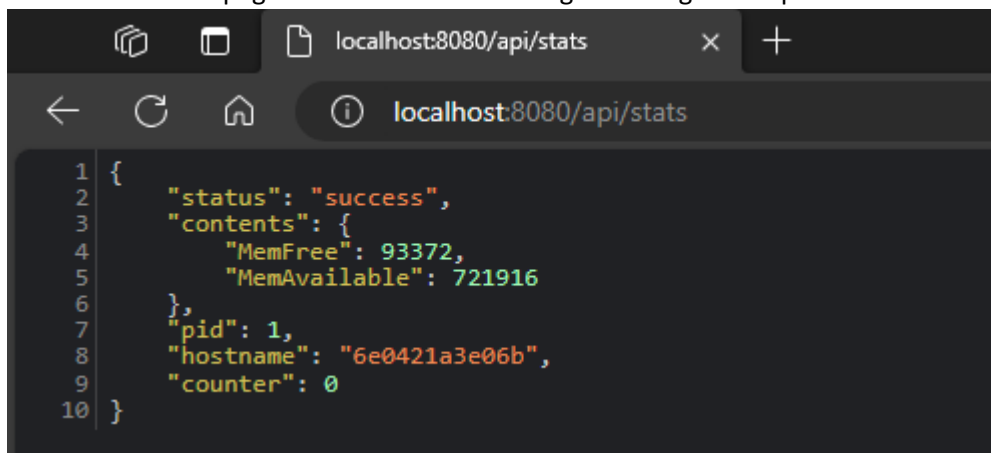
We will continue from the previous challenge where we successfully built our full stack application. First, let us make a few observations. Let us visit “http://localhost:8080/api/stats” to observe the output like so:



```
1 {
2   "status": "success",
3   "contents": {
4     "MemFree": 115880,
5     "MemAvailable": 743904
6   },
7   "pid": 1,
8   "hostname": "6e0421a3e06b",
9   "counter": 0
10 }
```

Notice the hostname.

Let us refresh the page and see the outcome again. Doing so will present us with something like this:



```
1 {
2   "status": "success",
3   "contents": {
4     "MemFree": 93372,
5     "MemAvailable": 721916
6   },
7   "pid": 1,
8   "hostname": "6e0421a3e06b",
9   "counter": 0
10 }
```

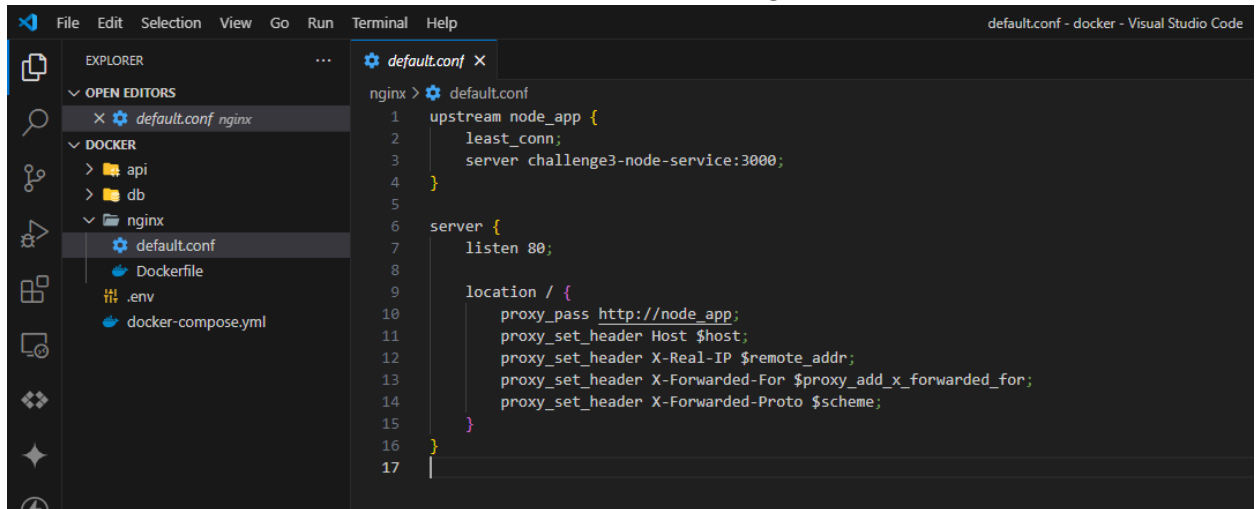
Notice the hostname stayed the same. This is because we are only running one instance of our “challenge3-node-service”. Now the goal in this challenge is to scale to three node instances. To start, we need to adjust certain files.

1. We will edit our “docker-compose.yml” file like so:

```
1 version: '3'
2
3 services:
4   challenge3-nginx:
5     build: ./nginx
6     ports:
7       - "8080:80"
8     depends_on:
9       - challenge3-node-service
10
11   challenge3-node-service:
12     build: ./api
13     environment:
14       DB_HOST: ${MYSQL_HOST}
15       DB_USERNAME: ${MYSQL_USER}
16       DB_PASSWORD: ${MYSQL_PASSWORD}
17       DB_DATABASE: ${MYSQL_DATABASE}
18       DB_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
19     volumes:
20       - ./api:/app
21       - /app/node_modules
22     # ports:
23     #   - "3000:3000"
24     depends_on:
25       - challenge3-db
26
27   challenge3-db:
28     build: ./db
29     environment:
30       MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
31       MYSQL_DATABASE: ${MYSQL_DATABASE}
32       MYSQL_USER: ${MYSQL_USER}
33       MYSQL_PASSWORD: ${MYSQL_PASSWORD}
34     volumes:
35       - db_data:/var/lib/mysql
36
37 volumes:
38   db_data:
```

The only thing we changed is to comment out the “ports” directive. Let me explain why: In a typical setup, we would have a reverse proxy (like Nginx which we are using) in front of our application instances, and only the reverse proxy would bind to a port on our host machine. The application instances would only bind to ports inside the Docker network where there’s no conflict. However, in our previous version of the “docker-compose.yml” file, we were mapping port 3000 of our host machine to port 3000 of the “challenge3-node-service” container. This works fine because we had only one instance of “challenge3-node-service”, but it causes a conflict when we try to scale to multiple instances because multiple services will be trying to use the same port in our host machine which will cause a conflict. We resolved this by removing the “ports” directive from the “challenge3-node-service” service definition in our “docker-compose.yml” file. This will stop the service from trying to bind to port 3000 on our host machine. Now, when we scale our “challenge3-node-service” to 3 instances, they will each bind to port 3000 inside their own Docker network, and Nginx will be able to proxy requests to them without any port conflicts.

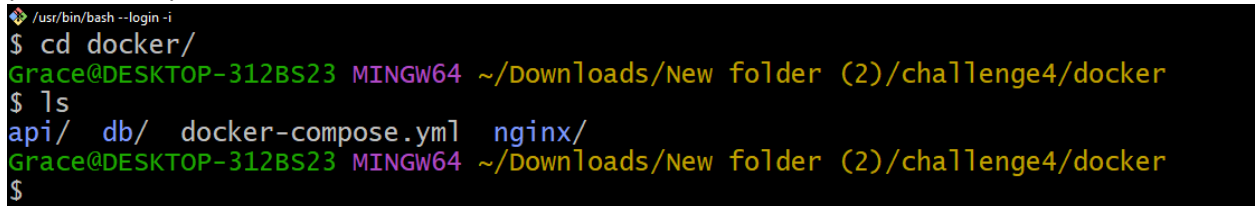
2. The second file we have to edit is our “default.conf” file in the nginx folder like so:



The screenshot shows the Visual Studio Code interface. On the left, the Explorer pane shows a project structure with folders 'api', 'db', and 'nginx'. Inside 'nginx', there is a file 'default.conf' which is selected. The main editor area shows the content of 'default.conf' with the following code:

```
1 upstream node_app {
2     least_conn;
3     server challenge3-node-service:3000;
4 }
5
6 server {
7     listen 80;
8
9     location / {
10        proxy_pass http://node_app;
11        proxy_set_header Host $host;
12        proxy_set_header X-Real-IP $remote_addr;
13        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
14        proxy_set_header X-Forwarded-Proto $scheme;
15    }
16 }
17
```

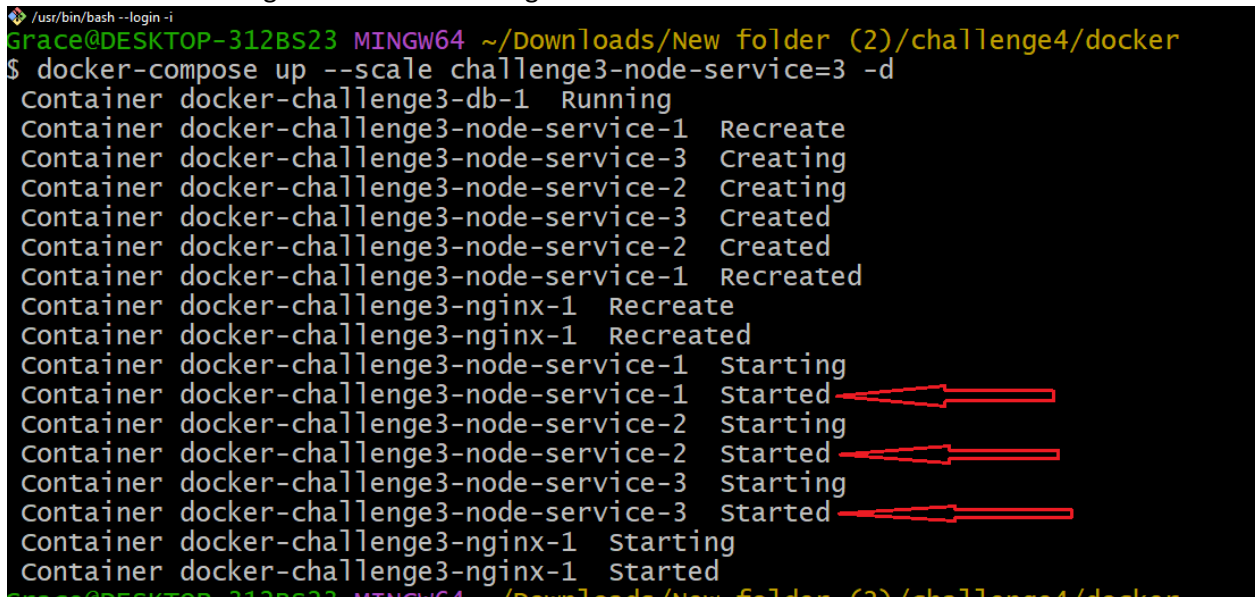
3. Now we will rerun the docker-compose command but with a slight variation. But first, ensure you’re still in your docker folder. You can run “ls” to confirm like so:



The screenshot shows a terminal window with the following commands and output:

```
/usr/bin/bash --login -i
$ cd docker/
Grace@DESKTOP-312BS23 MINGW64 ~/Downloads/New folder (2)/challenge4/docker
$ ls
api/ db/ docker-compose.yml nginx/
Grace@DESKTOP-312BS23 MINGW64 ~/Downloads/New folder (2)/challenge4/docker
$
```

4. Now the new docker-compose command to run is “docker-compose up --scale challenge3-node-service=3 -d”. This should create three instances of “challenge3-node-service”. Your terminal should look something like this after running this command.



The screenshot shows a terminal window with the following output after running the command:

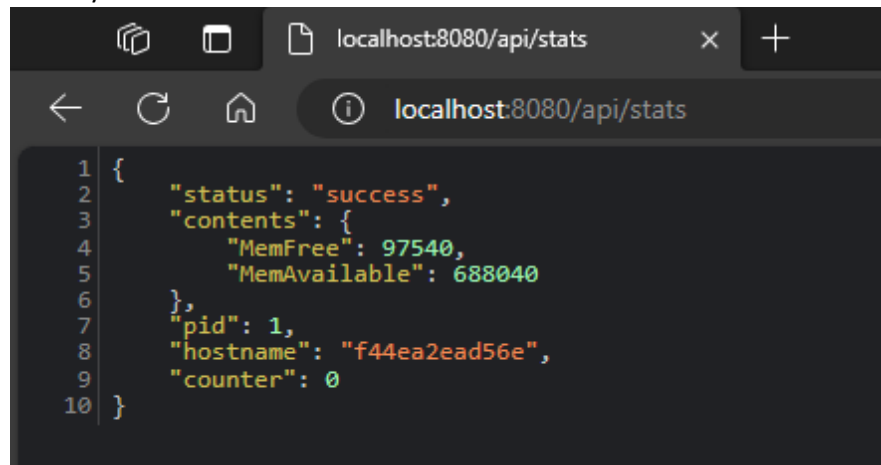
```
/usr/bin/bash --login -i
Grace@DESKTOP-312BS23 MINGW64 ~/Downloads/New folder (2)/challenge4/docker
$ docker-compose up --scale challenge3-node-service=3 -d
Container docker-challenge3-db-1 Running
Container docker-challenge3-node-service-1 Recreate
Container docker-challenge3-node-service-3 Creating
Container docker-challenge3-node-service-2 Creating
Container docker-challenge3-node-service-3 Created
Container docker-challenge3-node-service-2 Created
Container docker-challenge3-node-service-1 Recreated
Container docker-challenge3-nginx-1 Recreate
Container docker-challenge3-nginx-1 Recreated
Container docker-challenge3-node-service-1 Starting
Container docker-challenge3-node-service-1 Started
Container docker-challenge3-node-service-2 Starting
Container docker-challenge3-node-service-2 Started
Container docker-challenge3-node-service-3 Starting
Container docker-challenge3-node-service-3 Started
Container docker-challenge3-nginx-1 Starting
Container docker-challenge3-nginx-1 Started
Grace@DESKTOP-312BS23 MINGW64 ~/Downloads/New folder (2)/challenge4/docker
```

5. Now let us run “docker-compose ps” in our terminal.

```
container docker-challenge3-nginx-1 starting
Container docker-challenge3-nginx-1 started
Grace@DESKTOP-312BS23 MINGW64 ~/Downloads/New folder (2)/challenge4/docker
$ docker-compose ps
NAME                                IMAGE                                COMMAND                                SERVICE            CREATED
STATUS    PORTS
docker-challenge3-db-1              docker-challenge3-db                "docker-entrypoint.s..." challenge3-db        17 minutes ago
Up 17 minutes 3306/tcp
docker-challenge3-nginx-1           docker-challenge3-nginx              "/docker-entrypoint..." challenge3-nginx      46 seconds ago
Up 40 seconds 0.0.0.0:8080->80/tcp
docker-challenge3-node-service-1    docker-challenge3-node-service       "docker-entrypoint.s..." challenge3-node-service 57 seconds ago
Up 43 seconds 3000/tcp
docker-challenge3-node-service-2    docker-challenge3-node-service       "docker-entrypoint.s..." challenge3-node-service 57 seconds ago
Up 42 seconds 3000/tcp
docker-challenge3-node-service-3    docker-challenge3-node-service       "docker-entrypoint.s..." challenge3-node-service 57 seconds ago
Up 41 seconds 3000/tcp
Grace@DESKTOP-312BS23 MINGW64 ~/Downloads/New folder (2)/challenge4/docker
$
```

Notice we now have “docker-challenge3-node-service-1”, “docker-challenge3-node-service-2”, and “docker-challenge3-node-service-3”, where previously we had only had “docker-challenge3-node-service-1”. This means we have three instances running. Also, observe they are all running successfully.

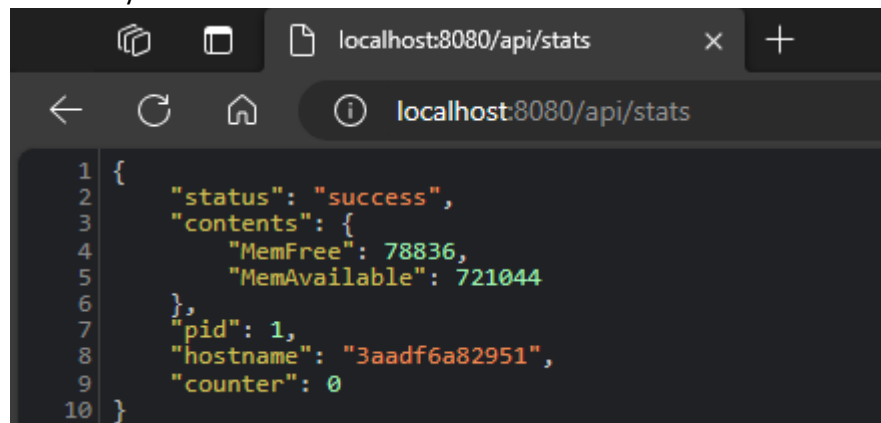
6. Now we can visit “http://localhost:8080/api/stats” again to observe the output.
- a. First try:



```
1 {
2   "status": "success",
3   "contents": {
4     "MemFree": 97540,
5     "MemAvailable": 688040
6   },
7   "pid": 1,
8   "hostname": "f44ea2ead56e",
9   "counter": 0
10 }
```

Notice the hostname. Let us refresh the page again.

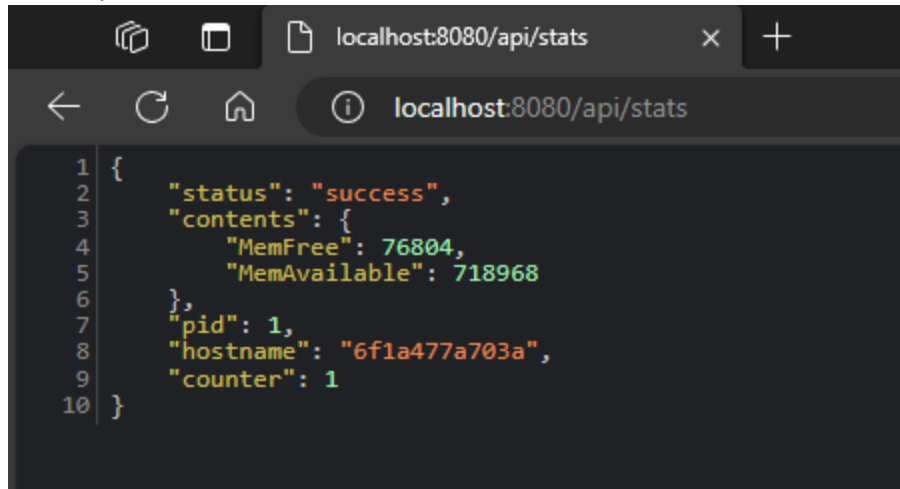
- b. Second try:



```
1 {
2   "status": "success",
3   "contents": {
4     "MemFree": 78836,
5     "MemAvailable": 721044
6   },
7   "pid": 1,
8   "hostname": "3aadf6a82951",
9   "counter": 0
10 }
```

Notice the hostname is different this time, unlike before.

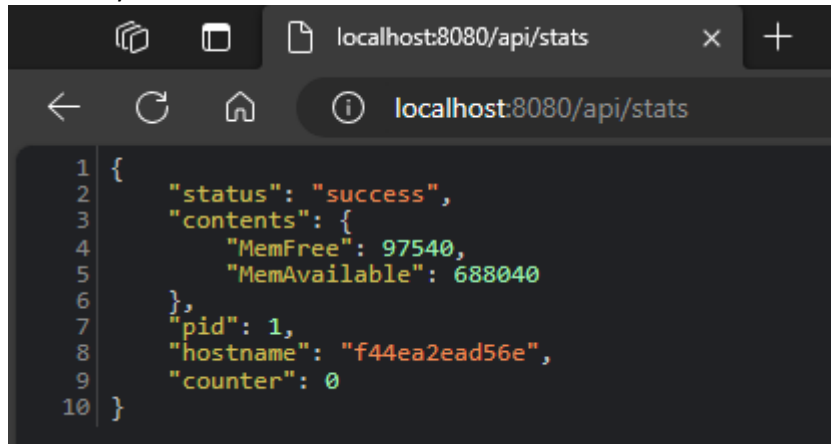
c. Third try:



```
1 {
2   "status": "success",
3   "contents": {
4     "MemFree": 76804,
5     "MemAvailable": 718968
6   },
7   "pid": 1,
8   "hostname": "6f1a477a703a",
9   "counter": 1
10 }
```

Notice the hostname is again different

d. Fourth try:



```
1 {
2   "status": "success",
3   "contents": {
4     "MemFree": 97540,
5     "MemAvailable": 688040
6   },
7   "pid": 1,
8   "hostname": "f44ea2ead56e",
9   "counter": 0
10 }
```

Notice the hostname now has the same value as it was when we first loaded the page.

What does this mean? In summary, there is an algorithm called “Round-Robin”, and it is a CPU scheduling or load balancing algorithm that distributes client requests across multiple servers in a cyclic manner. In the context of our setup with three Node.js instances, “docker-challenge3-node-service-1, 2, and 3” and one Nginx server (challenge3-nginx), round-robin would work as follows:

- i. The first incoming request is forwarded to the first Node.js instance.
- ii. The next request is forwarded to the second Node.js instance.
- iii. The third request is forwarded to the third Node.js instance.
- iv. The fourth request goes back to the first Node.js instance, and the cycle repeats.

This way, each Node.js instance gets an equal share of the load. This is the default behavior of Nginx when multiple servers are specified in an upstream block like we did when we modified our “default.conf” file in the nginx folder.

There are a few benefits to running multiple instances of a service (a practice often referred to as horizontal scaling) instead of one:

1. **Improved Availability:** If one instance goes down for any reason, the other instances can continue to handle requests.
2. **Load Balancing:** Multiple instances can share the load of incoming network traffic. When a lot of users are making requests to your application, having multiple instances can help ensure that the load is distributed, and no single instance becomes a bottleneck.
3. **Fault Isolation:** If an error occurs in one instance, it won't affect the others. This can be particularly useful when deploying new code. If there's an issue with the new code, only the instance(s) running the new code will be affected.
4. **Increased Capacity:** More instances mean your application can handle more concurrent requests, which can be particularly useful for applications expecting high levels of traffic.
5. **Scalability:** It's easier to scale out (adding more instances) than to scale up (increasing the resources of a single instance). You can quickly add more instances as demand increases, and remove them when demand decreases.

You can read more about the Round Robin algorithm, scaling to multiple instances, and more with the links below:

- [Round Robin Scheduling Algorithm with Example \(guru99.com\)](https://guru99.com/round-robin-scheduling-algorithm-with-example/)
- [Scaling with Docker Compose: A Beginner's Guide \(appsdeveloperblog.com\)](https://appsdeveloperblog.com/scaling-with-docker-compose-a-beginners-guide/)
- [Use Docker Compose | Docker Docs](https://docs.docker.com/compose/)

Congratulations on reaching this far! You have successfully built and scaled a full stack application complete with a web server, node application, and database successfully! Cheers to you!