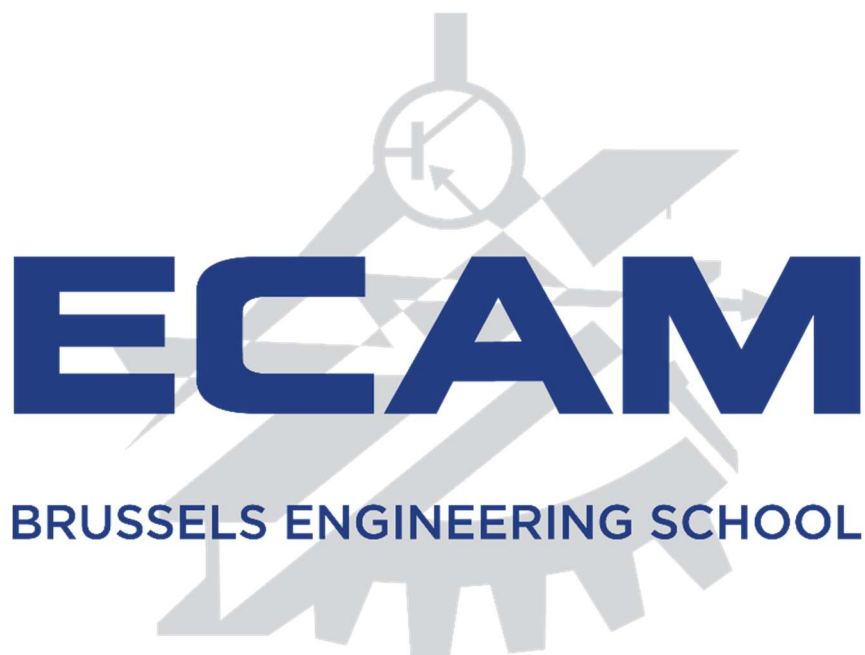


# Virtualization with NodeJS



Course:	5eiad50 Distributed Systems
Academic year:	2022-2023
Professor:	Q. Lurkin
Student:	Nick KUIJPERS - 20324

## Summary

1. Introduction .....	3
2. API Server .....	4
3. Load Balancer .....	5
4. Libraries used.....	6
4.1. Express .....	6
4.2. Cluster .....	6
4.3. Nginx .....	6
4.4. Cors .....	7
4.5. Body-parser .....	7
4.6. EJS .....	7

## 1. Introduction

For the course scalable architecture, we were demanded to build a server capable of answering queries on the Mandelbrot set. The request will contain an complex number and the server will respond by indicating whether it is part of the Mandelbrot set. We were also asked to make a graphical interface that displays the whole Mandelbrot by making requests to our server.

A load balancer distributes incoming network traffic across a group of servers. The load balancer sits between the client and the server farm accepting incoming network and application traffic and distributing the traffic across multiple servers using various methods. The main purpose of a load balancer is to improve the availability and fault tolerance of an application by spreading the workload across multiple servers and to increase the performance of the application by distributing the workload across multiple resources.

Github : [https://github.com/KuijpersNick0/Mandelbrot\\_MA2](https://github.com/KuijpersNick0/Mandelbrot_MA2)

## 2. API Server

I used Node.js to develop my server and application. It is designed to be lightweight and efficient, and it is often used to build scalable network applications. But by default, Node.js is single-threaded, which means that it can only run a single task at a time. However, it can perform asynchronous I/O operations, which allows it to perform multiple tasks concurrently even though it is single-threaded.

In a single-threaded environment, if a task blocks the event loop (for example, if it performs a long-running computation or waits for an I/O operation to complete), the entire application will become unresponsive until the task completes. This can be a problem in applications that need to handle a high volume of concurrent requests or that need to perform time-consuming tasks.

To overcome this limitation, I used the cluster module. It allows you to create a cluster of worker processes that share a single HTTP server. When a cluster is created, the master process forks a number of (as much as you give it CPU's available) worker processes, which all share the same HTTP server. When a request is received by the server, the load balancer distributes it among the worker processes in the cluster, allowing them to share the workload and to make better use of available resources.

This solution was enough, but it also had its limitations. It only used my cores and could not access the resources of a different computer. If my system had more than one application server to respond to this would fail.

I thus started another solution with Nginx in a docker environment. I used Nginx as a proxy-server. Nginx sits on the front of my server pool and distributes the requests among the workers.

### 3. Load Balancer

As explained before, the cluster module allows you to create a cluster of child processes that all share the same port. This can be used to create a load-balanced network of processes that can handle a larger number of requests than a single process could handle.

By default, the cluster module uses the round-robin algorithm to distribute incoming connections to the child processes. This means that it will rotate through the child processes in the order they were created, assigning each incoming connection to the next child process in the list.

Because this was limited to one server, I used Nginx as a proxy server. To use Nginx as a load balancer, I configured it to listen on the port that clients will connect to (80) and to forward requests to one or more backend servers (3002, 3003, 3004).

The upstream directive in Nginx defines a group of backend servers and then I had to use the proxy\_pass directive to forward client requests to the backend servers. The upstream block can contain one or more server blocks, each specifying the address and port of a backend server.

## 4. Libraries used

### 4.1. Express

Express is a fast, minimalist web framework for Node.js. It provides a simple, easy-to-use API for creating web servers and applications, and is designed to be flexible and extensible.

### 4.2. Cluster

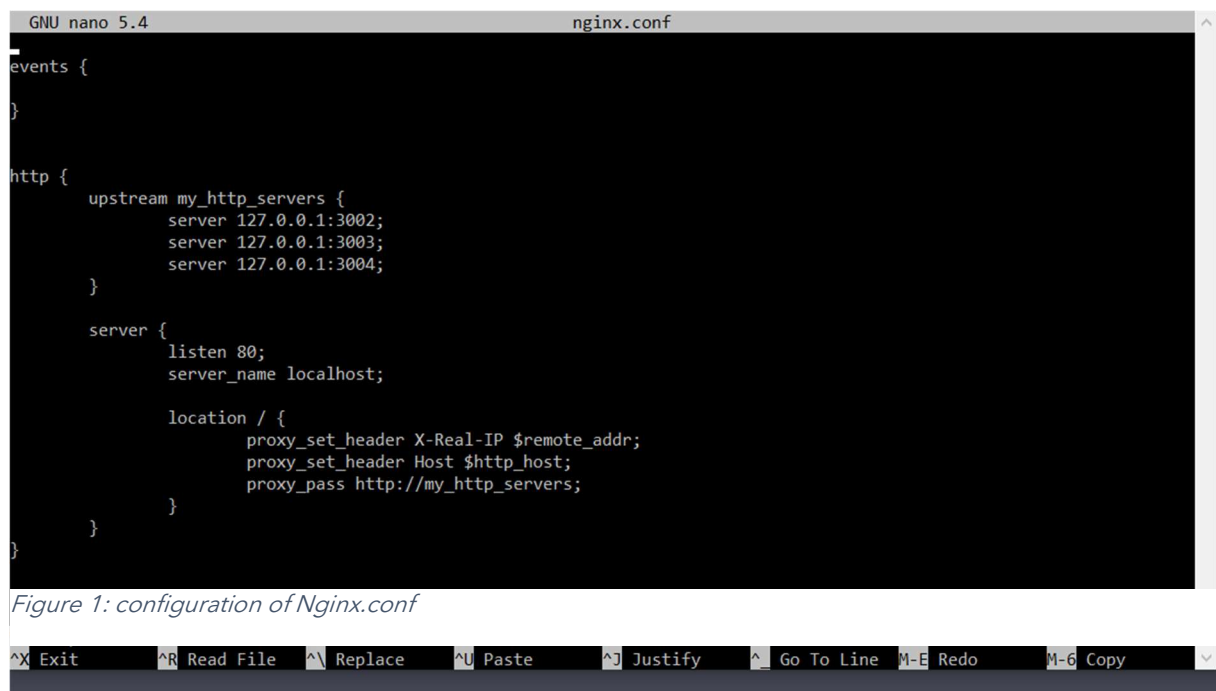
As explained before using this module we can launch NodeJS instances to each core of our system. A master process is listening on a port to accept client requests and distribute the workers.

### 4.3. Nginx

NGINX is a popular open-source web server and reverse proxy that can be used as a load balancer as explained before.

We used docker to start a Nginx container. We had to access the `/etc/nginx/nginx.conf` file to configure the Nginx as the proxy-server. To do this run in a command window : "docker container ls". This command will return all your containers with their id. We write down our container id for later. If Nano is installed we can directly run "docker exec -it <container\_name\_or\_id> nano /etc/nginx/nginx.conf". This was not our case so we first had to install a text editor (vi or else didn't work either). With "docker exec -it <container\_name\_or\_id> bash" we could connect to the docker. Then run : apt-get update and apt-get install nano. We could now access our file with nano using nano `/etc/nginx/nginx.conf` :

After the modification, don't forget to "nginx -s reload".



```

GNU nano 5.4                               nginx.conf
events {
}

http {
    upstream my_http_servers {
        server 127.0.0.1:3002;
        server 127.0.0.1:3003;
        server 127.0.0.1:3004;
    }

    server {
        listen 80;
        server_name localhost;

        location / {
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header Host $http_host;
            proxy_pass http://my_http_servers;
        }
    }
}

^X Exit  ^R Read File  ^\ Replace  ^U Paste  ^] Justify  ^_ Go To Line  M-E Redo  M-6 Copy

```

Figure 1: configuration of Nginx.conf

#### 4.4. Cors

CORS stands for Cross-Origin Resource Sharing. It allows us to relax the security applied to an API. This is done by bypassing the Access-Control-Allow-Origin headers, which specify which origins can access the API.

#### 4.5. Body-parser

The body-parser module is a middleware function that parses a request body of an HTTP request. It reads the request body, and then makes the request data available in the req.body property of the request object for my routes to use. It supports JSON format. Combined with Express it is very easy to create an API in NodeJs.

#### 4.6. EJS

EJS (Embedded JavaScript) is a simple templating language that lets you generate HTML, XML, or other text files by embedding JavaScript code in templates.