Designing and implementing a scalable distributed application

Bellihi Ismail

Submitted to
School of Electrical Engineering, Aalto University
Department of Communications and Networking
MOSA!C Lab, www.mosaic-lab.org

January 2020

Contents

List of Figures

1

Introduction

The development of web applications becomes so feasible due to the number of frameworks and even the application generators, but developing an application that ensures the availability and tolerance to failure with zero downtime. That is the challenge of modern applications. When developing the first version of an application, we often do not have any scalability issues. Moreover, using a distributed architecture slows down development.

Imagine that we have our server and we have a major of HTTP requests coming in all of a sudden and our server can't handle them at the same time. The second issue appears when we have a really computationally expensive endpoint and our API is doing some long tasks, like image processing or some other algorithms that are holding up a lot of client's HTTP requests that are coming in, because it's blocking them, so that is another issue with web applications.

Another issue is when our server goes down and we can't respond to any incoming request until we repair the server and restarting the server unfortunately that will take a lot of time. To resume the issues of functioning of web application in a High number of requests, hitting computationally slow endpoints and server failure. In this report we will discuss some techniques to increasing the availability and the scalability of web applications.

In the first part we will presenting some architectures, following by covering some concepts of implementations withing **nodejs** and **expressjs** framework.

2

Techniques and architectures

2.1 Load Balancer

process of spreading requests across multiple resources according to some metric Load balancing is an approach to distribute and spreading requests traffic to multiple resources according to some metric.

In high traffic apps, we can't rely on server handling every request. We run multiple application servers preferably on different machines and distribute our traffic among them to reduce response time and achieve higher availability.

The load balancer stays in between the client and application servers and decides on which server this request should go. The decision can be configured using different algorithms.

- **Round-robin** It's the default algorithm, consist of assigning the each request to the next node, in equal portions and in circular order, without priority.
- **Least Connection** The current request goes to the server that is servicing the least number of active sessions at the current time.
- **Weighted Response Time** This method dispatch the current request to the server, witch respond more faster. This allows the load to even out on the available server pool over time.

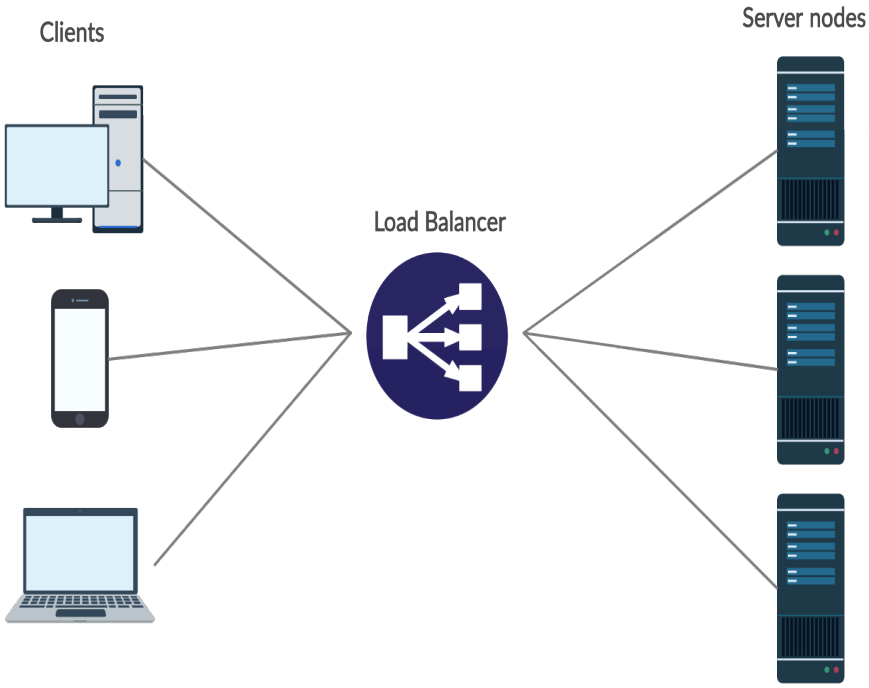


Figure 2.1: Load balancing schema

2.2 Cache node

This technique consist to setting up a server cache in order to speed up the response time of a server, and to avoid the fetching the data recently requested, the node will quickly return local cached data if it exists. If it is not in the cache, the requesting node will query the data from the source.

This concept used for reducing the bandwidth consumption. the node holds copies of data passing through it, for a limiting period. check the memory cache for each incoming request if it already requested we sent the cache data, in other case we send the request to the main server to retrieve the data from source, then we send it to the user, and we store this data in our cache to serving the next incoming requests.

2.3 Message Broker (RabbitMQ)

RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol. Node.js is in practice single-threaded, so a single instance of the process can only perform one action at a time, so imagine that one user send a data witch require a some computational time using a sing process force others wait for finishing, however when we use Message broker we can just take a request from the user put it in the queue with delayed if there are no resources available at that moment, and start with other requests.

3

Implementation

3.1 Tools and technologies

3.1.1 Nodejs

`NodeJs` is an open-source, JavaScript runtime environment that executes JavaScript code outside of a browser. Node.js helps in the execution of JavaScript code server-side. It's quite easy for developers to scale the applications in vertical as well as horizontal.

3.1.2 NGINX

`NGINX` Nginx is a web server which can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. The software was created by **Igor Sysoev** and first publicly released in 2004.

3.2 Solution

3.2.1 scaling a Node.js application

There are two approches for scaling a web application, the first is to scaling it Verticalaly by focusing on performances of single server performances, such as (CPU, RAM).

The second is horizontal scaling, consist to duplicating the application into many instances to manage a large number of incoming requests, this concept can be performed on a single multi-core machine or across different machines. In this solution I will cover the distribution on a single multi-core machine.

3.2.2 performance measurement with one single node

To measure the performance and the response time of my application I used **apache benchmarking tool (ab)**.

 # ab -p credentials.json -T application/json -c 10 -n 100 -l http://localhost:3000/users/signup

This command will test-load the server with 10 concurrent connections for 100 requests.

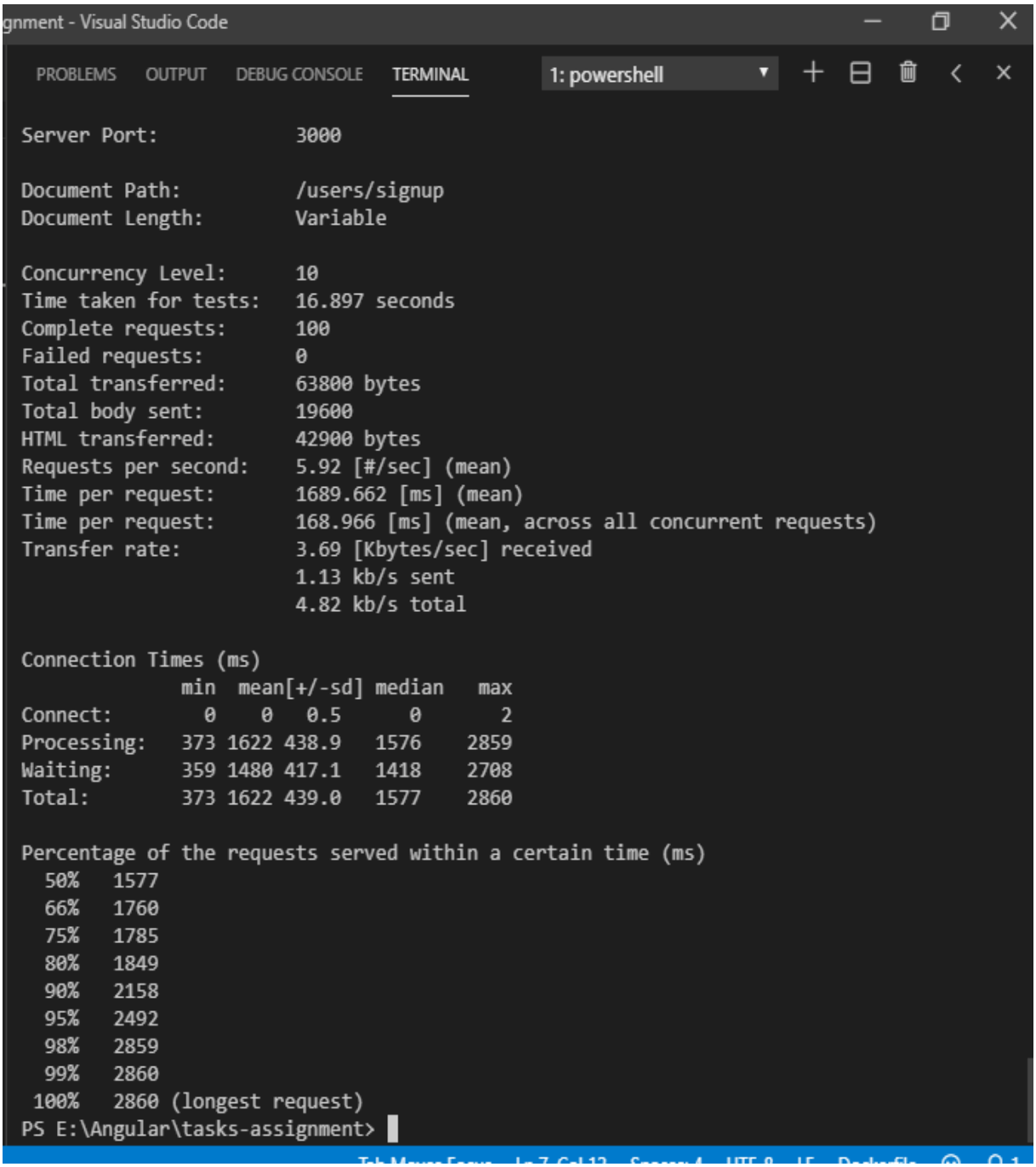


Figure 3.1: Performance of single mode

The single node server on my machine was able the handle **6** post request per second. this result will not be the same for different machine, this test is show

clearly how the server respond in single node. for example with single nodejs instance we have 6 request per second, and **1480 milliseconds** as mean of waiting time for the concurrent request.

3.2.3 performance measurement with distributed node instances

I used **PM2** the process manager for the JavaScript runtime Node.js, allows networked Node.js applications to be scaled accross all CPUs available. This greatly increases the performance and reliability of ourr applications, depending on the number of CPUs available. for example in my docker container I have two CPUs.

```
/usr/src/app # pm2 list
```

id	name	mode	↺	status	cpu	memory
0	index	cluster	0	online	0%	76.2mb
1	index	cluster	0	online	0%	79.6mb

```
/usr/src/app #
```

Figure 3.2: List of instances

after I configure the PM2 to duplicate the server in multi instance I tested again the performance of tow instances, I got this statistics.

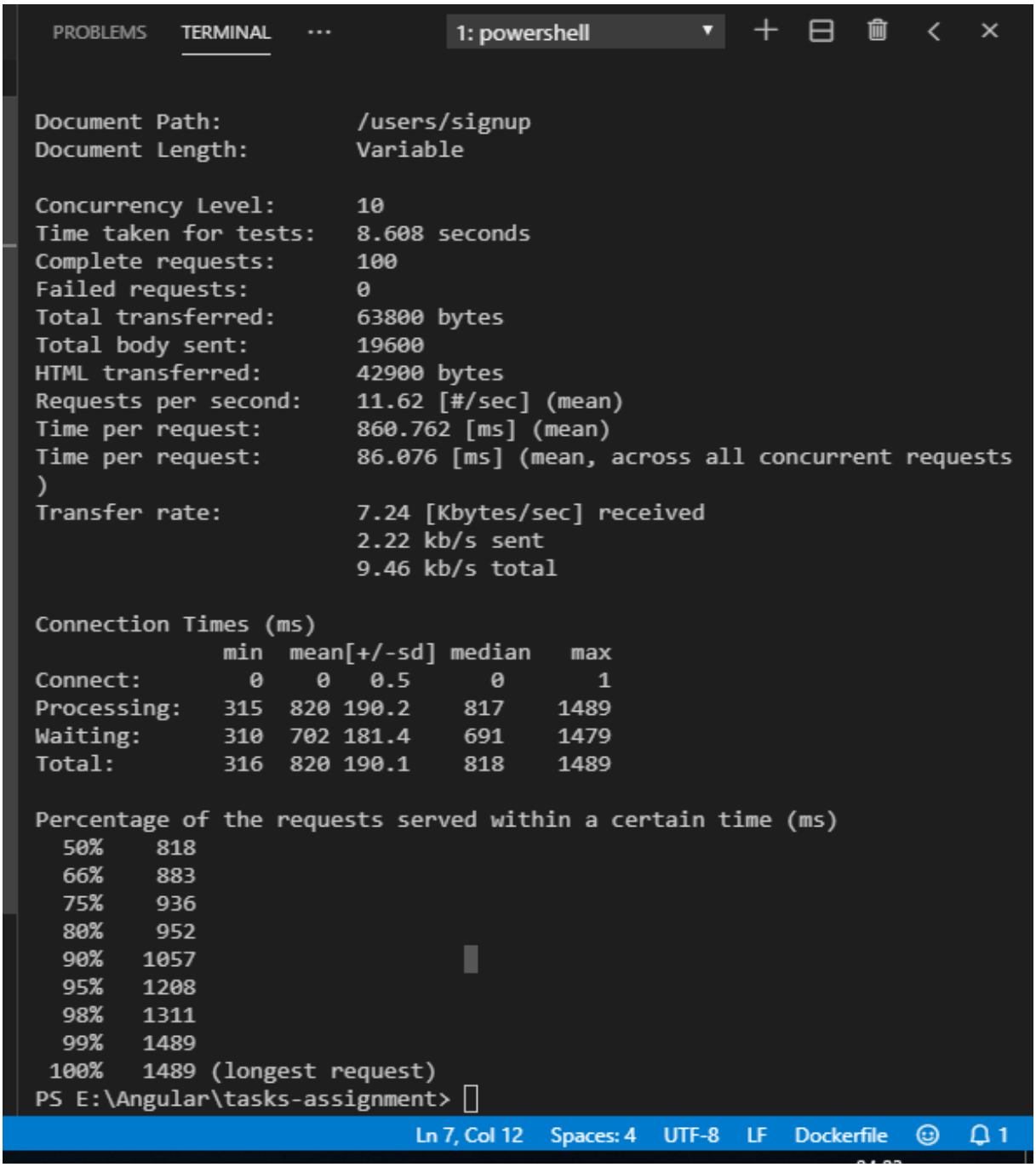


Figure 3.3: Performance of duplicate instances

The multi node was able to handle **12** post requests per second, more faster than mono node, also decrease the waiting time for the concurrent requests **702**

milliseconds.

3.2.4 other best practices for scaling the web application

Stateless authentication

For the user authentication there are many mechanisms, either using user session or using the authentication by tokens... .

In order to ensure the authentication of users, and for example, a user requests an authorization by sending his credentials we check his credentials and returning the result, then if we want to save the status of users as authenticated for the next requests we need to store his session in our server, so the problem of authenticate this user in all the instances of our load balancer, the trivial solution to this problem is to persist the user session in all instances, that is definitely not good from the performance point of view.

On the side of embracing a more efficient approach to stateless authentication, I used **JSON Web Tokens**.

The principle of JWT is when a user logs in, the server generates a token that is essentially a base64 encoding of a JSON object containing the payload, plus a signature obtained hashing that payload with a secret key owned by the server. The payload can contain data used to authenticate and authorize the user, for example, the user login. then once the user is logged in, each subsequent request will include the token, allowing the user to access routes, services, and resources that are permitted with that token after checking the token payload, so no need to store any information.

4

Conclusions

Designing efficient and distributed systems with fast access to lots of data and fast responding to large requests number is exciting, and there are lots of great tools that enable all kinds of new applications. I covered in this report just a few examples and few best practices for designing an auto-scalable web application using nodejs, but there are many more tools and techniques (PaaS) and there will only continue to be more innovation in the space.

5

Annex

5.1 References

1. Good practices for high-performance and scalable Node.js applications (medium)
2. https://mongoosejs.com/docs/guide.html
3. https://pm2.keymetrics.io/
4. https://www.nginx.com/

5.2 Documentation

1. source code available on **https://github.com/isBellihi/tasks-assignment**

This document describe how deploy and start using this application in your local machine

Download and clone the repository from github

```
Clone the project in your local machine
```

```
$ git clone https://github.com/isBellihi/tasks-assignment.git (https://github.com/isBellihi/tasks-assignment.git)
```

```
Access to the project folder
```

```
$ cd tasks-assignment
```

We supposed that you have docker already installed otherwise install it

```
https://www.docker.com/products/docker-desktop (https://www.docker.com/products/docker-desktop)
```

Deploy and Launch application with docker compose

```
$ docker-compose up
```

After docker-compose command finish, pulling and running the dependencies

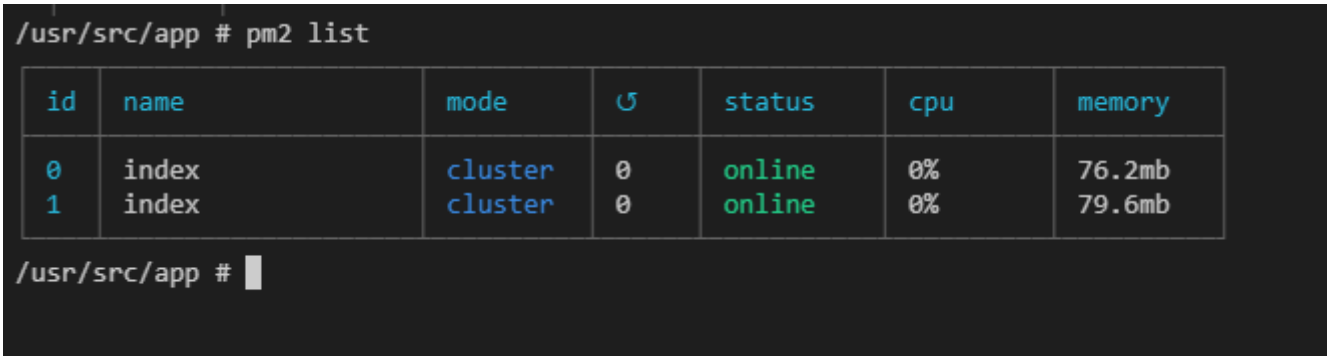
You will be able now to run the application and interact with the API using postman

Run a command and intercating with a running container

```
$ docker exec -it tasks-assignment /bin/sh -c "[ -e /bin/bash ] && /bin/bash || /bin/sh"
```

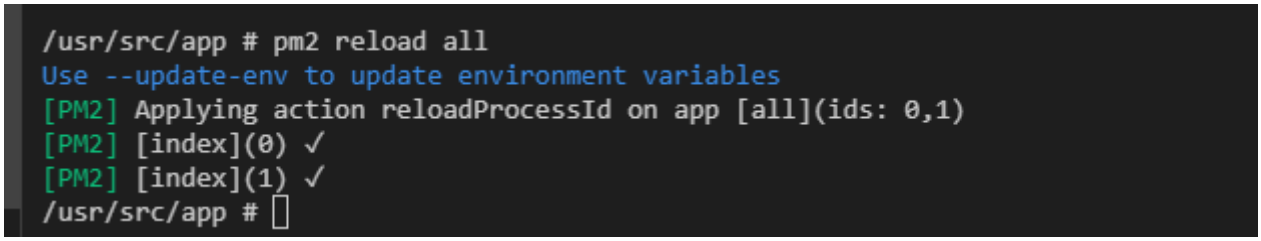
list the running instances

```
$ pm2 list
```



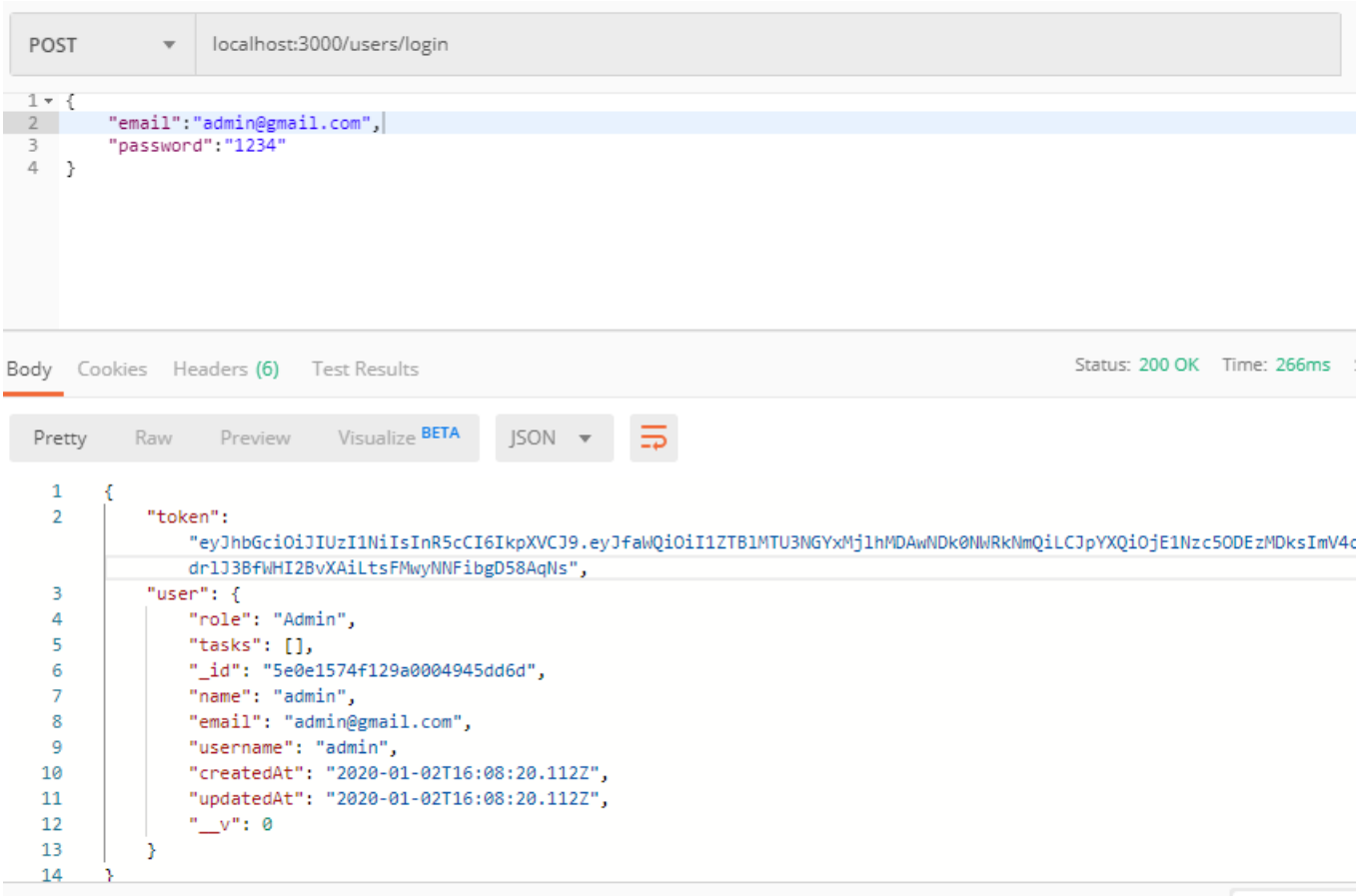
reload all instances with zero downtime

```
$ pm2 reload all
```

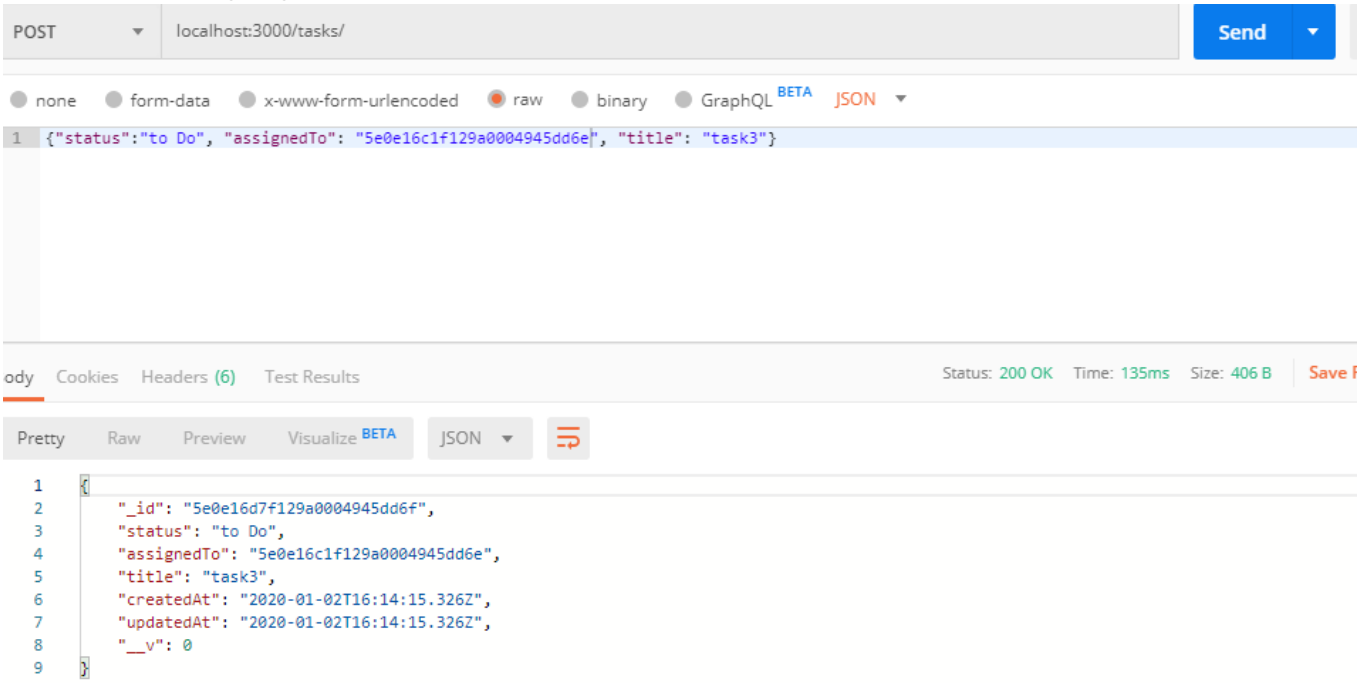


Illustarations

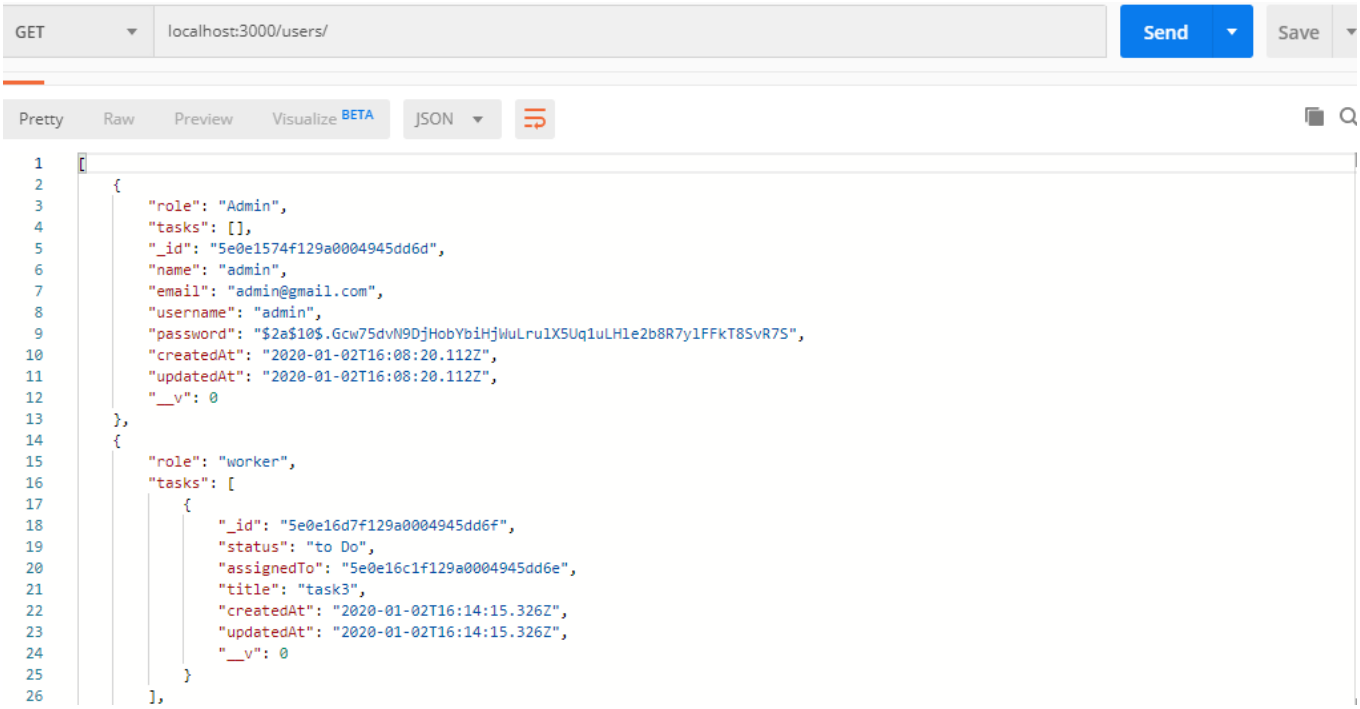
signing in


POST
localhost:3000/users/login
{
"email":"admin@gmail.com",
"password":"1234"
}
Body
Cookies
Headers (6)
Test Results
Status: 200 OK
Time: 266ms
Pretty
Raw
Preview
Visualize BETA
JSON
{
"token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI1ZTBlMTU3NGYxMjlhMDAwNDk0NWRkNmQiLCJpYXQiOjE1Nzc5ODEzMDksImV4c
drlJ3BfWHI2BvXAiLtsFMwyNNFibgD58AqNs",
"user": {
"role": "Admin",
"tasks": [],
"_id": "5e0e1574f129a0004945dd6d",
"name": "admin",
"email": "admin@gmail.com",
"username": "admin",
"createdAt": "2020-01-02T16:08:20.112Z",
"updatedAt": "2020-01-02T16:08:20.112Z",
"__v": 0
}
}

Add task and assigning it to a worker



get All users with their tasks




```
[
    {
        "role": "Admin",
        "tasks": [],
        "_id": "5e0e1574f129a0004945dd6d",
        "name": "admin",
        "email": "admin@gmail.com",
        "username": "admin",
        "password": "$2a$10$.Gcw75dvN9DjHobYbiHjWuLrulX5Uq1uLH1e2b8R7y1FFkT8SvR7S",
        "createdAt": "2020-01-02T16:08:20.112Z",
        "updatedAt": "2020-01-02T16:08:20.112Z",
        "__v": 0
    },
    {
        "role": "worker",
        "tasks": [
            {
                "_id": "5e0e16d7f129a0004945dd6f",
                "status": "to Do",
                "assignedTo": "5e0e16c1f129a0004945dd6e",
                "title": "task3",
                "createdAt": "2020-01-02T16:14:15.326Z",
                "updatedAt": "2020-01-02T16:14:15.326Z",
                "__v": 0
            }
        ],
```


get All tasks with the status (toDo, Doing, Done)


```
[
    {
        "_id": "5e0e16d7f129a0004945dd6f",
        "status": "to Do",
        "assignedTo": {
            "role": "worker",
            "tasks": [
                "5e0e16d7f129a0004945dd6f"
            ],
            "_id": "5e0e16c1f129a0004945dd6e",
            "name": "worker1",
            "email": "worker1@gmail.com",
            "username": "worker1",
            "password": "$2a$10$EM05YbpVyV/fDQH5PWXgn.nBt4VD4raU9wjVzxU3196FO/7HmY1Oa",
            "createdAt": "2020-01-02T16:13:53.124Z",
            "updatedAt": "2020-01-02T16:14:15.369Z",
            "__v": 1
        },
        "title": "task3",
        "createdAt": "2020-01-02T16:14:15.326Z",
        "updatedAt": "2020-01-02T16:14:15.326Z",
        "__v": 0
    }
]
```