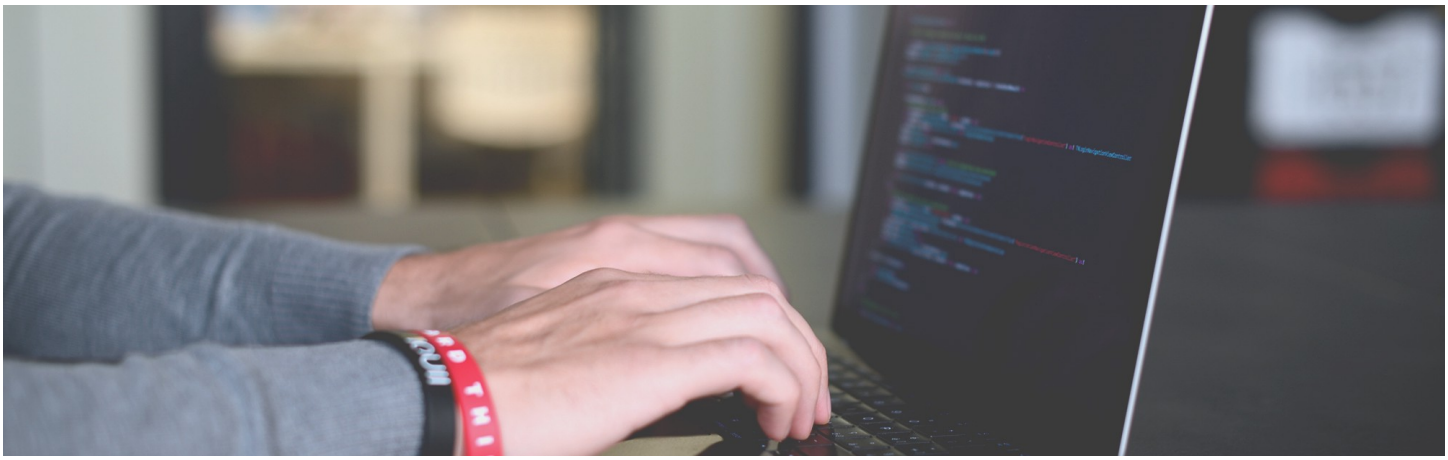
Good practices for high-performance and scalable Node.js applications [Part 1/3]

virgafox
Jan 12, 2018 · 5 min read



In this series of 3 articles we will cover some good practices about developing a Node.js web back-end application.

The series will not be a tutorial about Node, all the things you will read are intended for developers already familiar with the basics of Node.js and are looking for some hints about improving their architectures.

The main focus will be about efficiency and performance, in order to obtain the best result with less resources.

One way to improve the throughput of a web application is to scale it, instantiate it multiple times balancing the incoming connection between the multiple instances, so this first article will be about **how to horizontally scale a Node.js application**, on multiple cores or on multiple machines.

When you scale up, you have to be careful about different aspects of your application, from the state to the authentication, so the second article will cover some **things you must consider** when scaling up a Node.js application.

Over the mandatories ones, there are some **good practices you can address** that will be covered in the third article, like splitting api and worker processes, the adoption of priority queues, the management of periodic jobs like cron processes, that are not intended to run N times when you scale up to N processes/machines.

Chapter 1 — Horizontally scaling a Node.js application

Horizontal scaling is about duplicating your application instance to manage a larger number of incoming connections. This action can be performed on a single multi-core machine or across different machines.

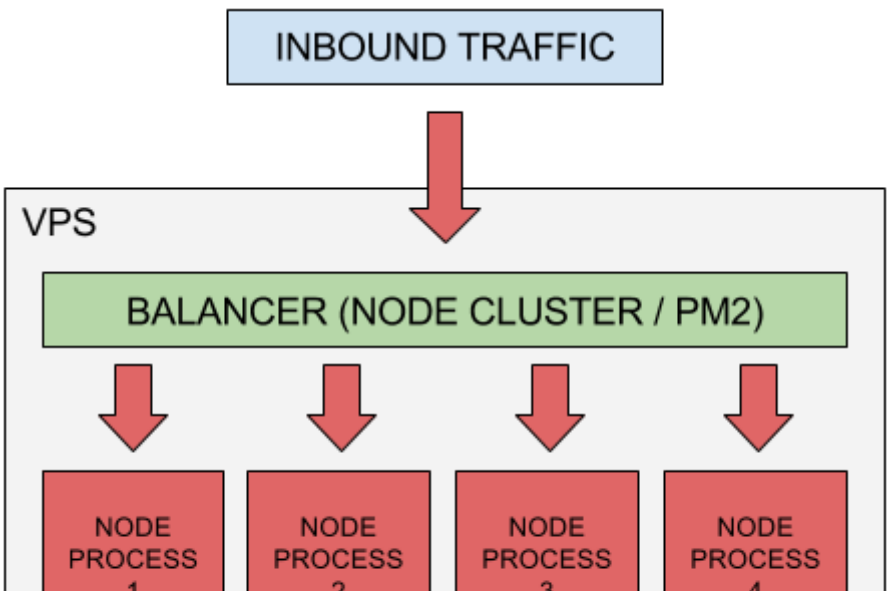
Vertical scaling is about increasing the single machine performances, and it do not involve particular work on the code side.

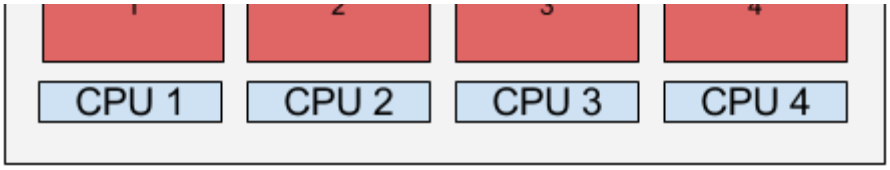
Multiple processes on same machine

One common way to increase the throughput of your application is to spawn one process for each core of your machine. By this way the already efficient “concurrency” management of requests in Node.js (see “event driven, non-blocking I/O”) can be multiplied and parallelized.

It is probably not clever to spawn a number of processes bigger than the number of cores, because at the lower level the OS will likely balance the CPU time between those processes.

There are different strategies for scaling on a single machine, but the common concept is to have multiple processes running on the same port, with some sort of internal load balancing used to distribute the incoming connections across all the processes/cores.





The strategies described below are the standard Node.js **cluster mode** and the automatic, higher-level **PM2 cluster** functionality.

Native cluster mode

The native Node.js cluster module is the basic way to scale a Node app on a single machine (https://Node.js.org/api/cluster.html). One instance of your process (called “master”) is the one responsible to spawn the other child processes (called “workers”), one for each core, that are the ones that runs your application. The incoming connections are distributed following a round-robin strategy across all the workers, that exposes the service on the same port.

The main drawback of this approach is the necessity to manage inside the code the difference between master and worker processes manually, typically with a classic if-else block, without the ability to easily modify the number of processes on-the-fly.

The following example is taken from the official documentation:

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {

 console.log(`Master ${process.pid} is running`);

 // Fork workers.
 for (let i = 0; i < numCPUs; i++) {
  cluster.fork();
 }

 cluster.on('exit', (worker, code, signal) => {
  console.log(`worker ${worker.process.pid} died`);
 });

} else {

 // Workers can share any TCP connection
 // In this case it is an HTTP server
```

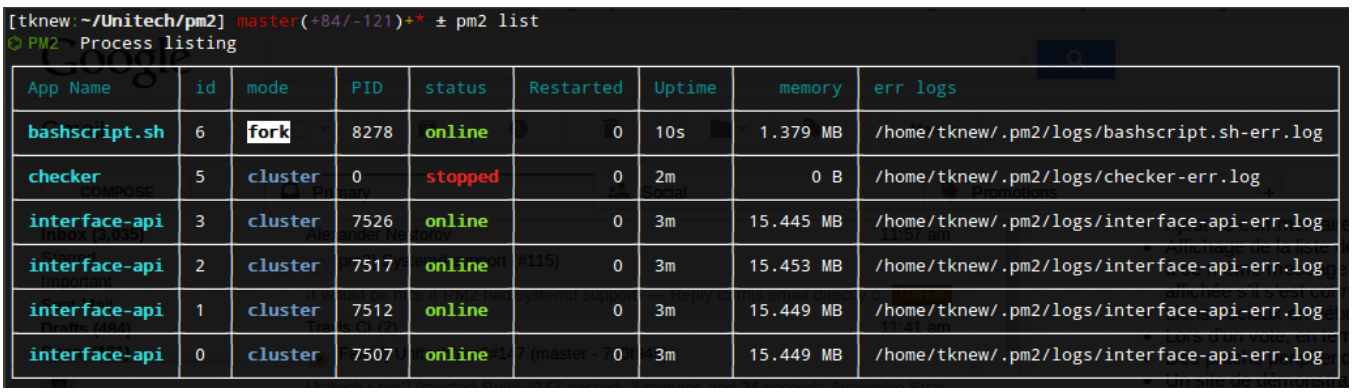
```
  http.createServer((req, res) => {
   res.writeHead(200);
   res.end('hello world\n');
  }).listen(8000);

  console.log(`Worker ${process.pid} started`);

}
```

cluster.js hosted with ❤ by GitHub view raw

PM2 Cluster mode

If you are using PM2 as your process manager (I suggest you to), there is a magic cluster feature that let you scale your process across all the cores without worrying about the cluster module. The PM2 daemon will cover the role of the "master" process, and it will spawn N processes of your application as workers, with round-robin balancing.

By this way you simply write your application as you would do for single-core usage (with some cautions that we'll cover in next article), and PM2 will care about the multi-core part.



```
[tknew:~/Unitech/pm2] master(+84/-121)+* ± pm2 list
PM2 Process listing
```

App Name	id	mode	PID	status	Restarted	Uptime	memory	err logs
bashscript.sh	6	fork	8278	online	0	10s	1.379 MB	/home/tknew/.pm2/logs/bashscript.sh-err.log
checker	5	cluster	0	stopped	0	2m	0 B	/home/tknew/.pm2/logs/checker-err.log
interface-api	3	cluster	7526	online	0	3m	15.445 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	2	cluster	7517	online	0	3m	15.453 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	1	cluster	7512	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	0	cluster	7507	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log

Once your application is started in cluster mode, you can adjust the number of instances on-the-fly using "pm2 scale", and perform "0-second-downtime" reloads, where the processes are restarted in series in order to have always at least one process online.

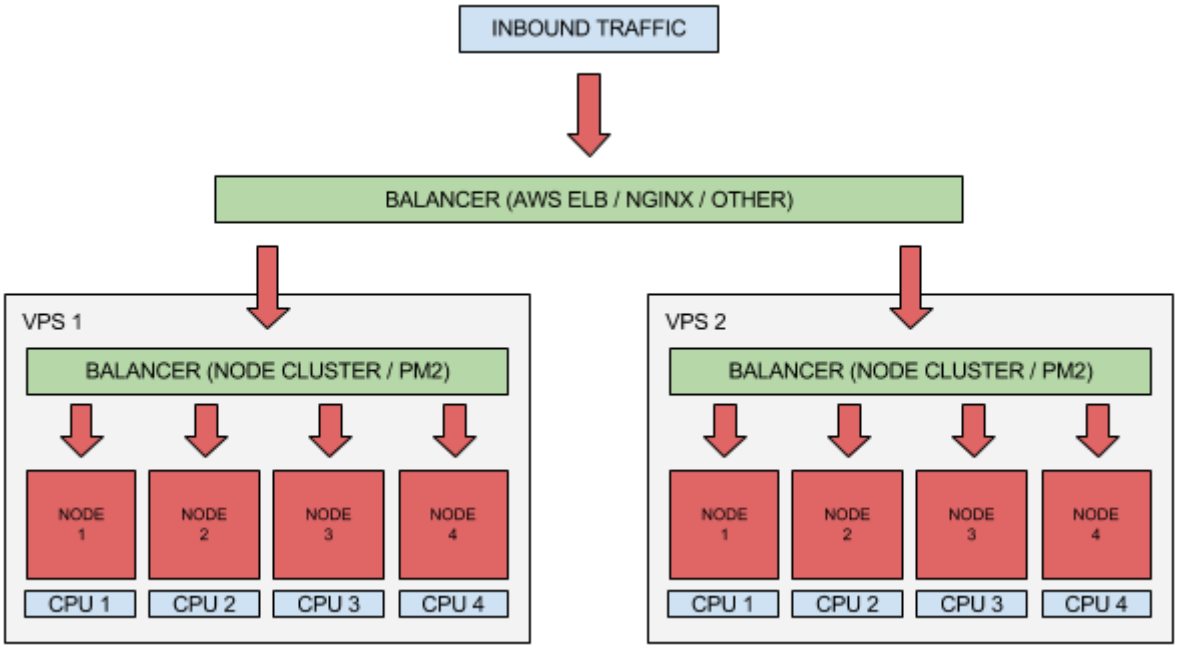
As a process manager, PM2 will also take care of restarting your processes if they crash like many other useful things you should consider when running node in production.

If you need to scale even further, you'll probably need to deploy more machines.

Multiple machines with network load balancing

The main concept in scaling across multiple machines is similar to scaling on multiple cores, there are multiple machines, each one running one or more processes, and a balancer to redirect traffic to each machine.

Once the request is sent to a particular node, the internal balancer described in the previous paragraph sends the traffic to a particular process.



A network balancer can be deployed in different ways. If you use AWS to provision your infrastructure, a good choice is to use a managed load balancer like ELB (Elastic Load Balancer), because it supports useful features like auto-scaling, and it is easy to set up.

But if you want to do it old-school, you can deploy a machine and setup a balancer with NGINX by yourself. The configuration of a reverse proxy that points to an upstream is quite simple for this job. Below an example for the configuration:

```
http {

 upstream myapp1 {
   server srv1.example.com;
   server srv2.example.com;
   server srv3.example.com;
 }

 server {
   listen 80;
   location / {
    proxy_pass http://myapp1;
```

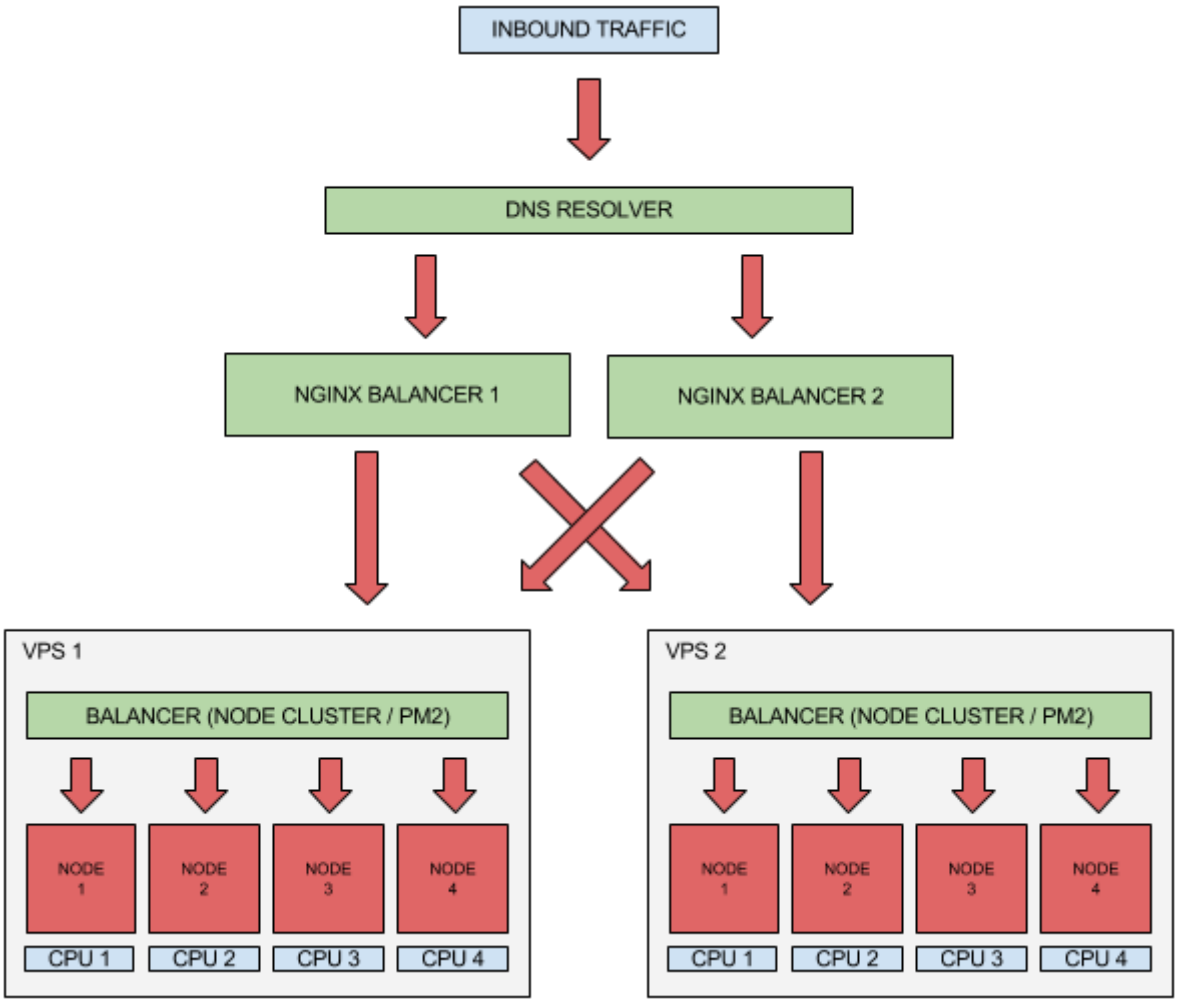
```
13      }
14    }
15
16  }
```

balancer.conf hosted with ❤️ by GitHub view raw

By this way the load balancer will be the only entrypoint of your application exposed to the outer world. If you worry about it being the single point of failure of your infrastructure, you can deploy multiple load balancers that points to the same servers.

In order to distribute the traffic between the balancers (each one with its own ip address), you can add multiple DNS "A" records to your main domain, so the DNS resolver will distribute the traffic between your balancers, resolving to a different IP address each time.

By this way you can achieve redundancy also on the load balancers.



Next steps

What we have seen here is how to scale a Node.js app at different levels in order to obtain the highest possible performance from your infrastructure, from single node, to multi node and multi balancer, but be careful: if you want to use your application in a multi-process environment, it must be prepared and ready for that, or you will incur in several problems and undesired behaviours.

In the next article we'll find out what to do to make your application scale-ready. You can find it here.

. . .

Clap as much as you like if you appreciate this post!

Nodejs DevOps JavaScript Developer Backend