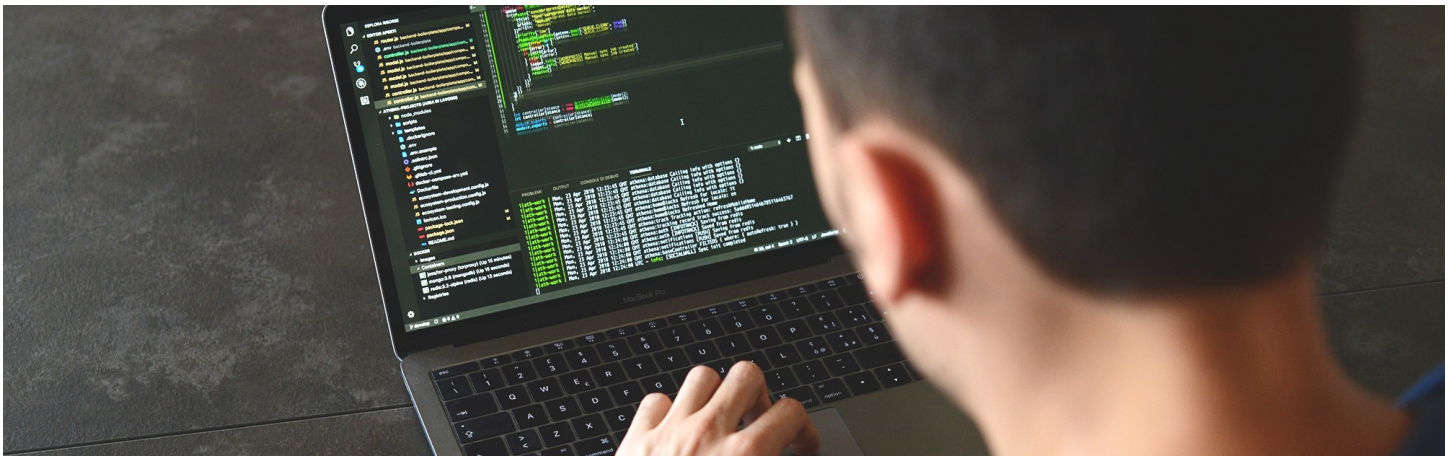
Good practices for high-performance and scalable Node.js applications [Part 3/3]

virgafox

Apr 23, 2018 · 5 min read



Chapter 3 — Additional good practices for efficiency and performance

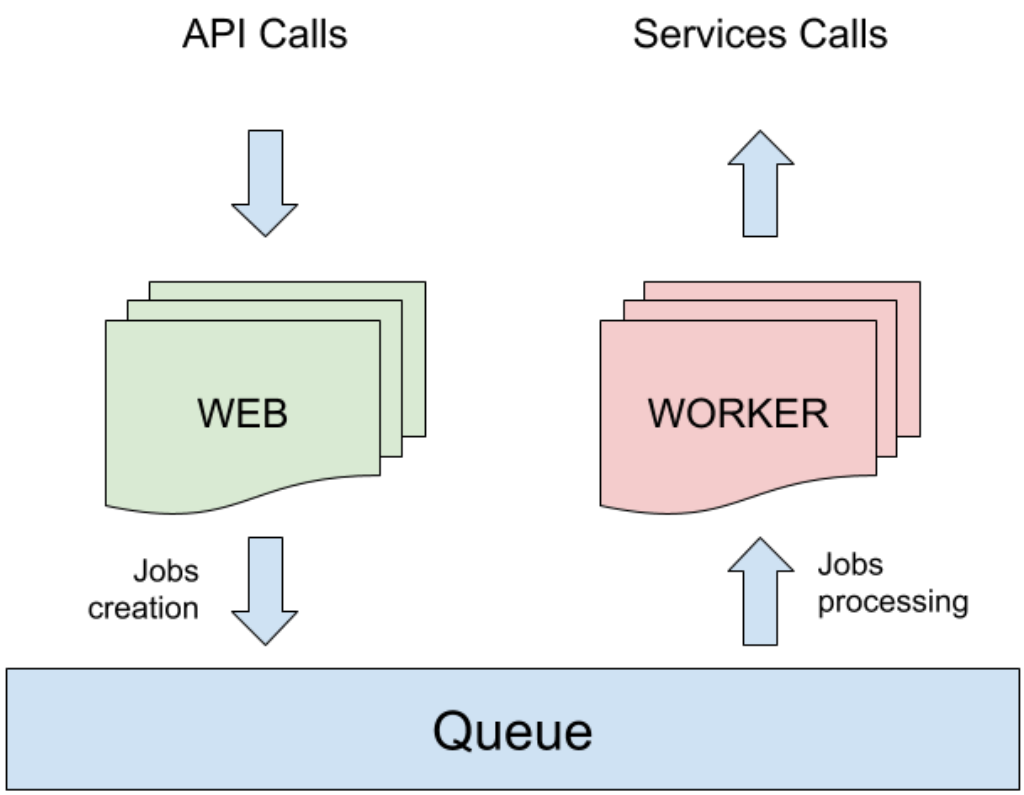
In the first two articles of this serie we saw how to scale a Node.js application and what to consider on the code side to make it behave as expected during this process. In this last article we'll cover some additional practices that can improve efficiency and performance a little further.

Web and Worker processes

As you probably know, **Node.js is in practice single-threaded**, so a single instance of the process can only perform one action at a time. During the life-cycle of a web application, **many different tasks** are performed: managing API calls, reading/writing to DB, communicating with external network services, execution of some sort of inevitable CPU intensive work etc.

Although you use asynchronous programming, delegating all these actions to the same process that responds to your API calls can be a quite inefficient approach.

A common pattern is based on the **separation of responsibilities** between two different kind of processes that compose your application, usually a **web** process and a **worker** one.



The web process is designed to mainly manage the **incoming network calls**, and to dispatch them as fast as possible. Any time a non-blocking task needs to be performed, like for example sending an email/notification, write a log, execute a triggered action which result is not necessary to answer the API call, the web process delegates the action to the worker one.

The **communication between web and worker** processes can be implemented in different ways. A common and efficient solution is a priority queue, like the one implemented in Kue, described in the next paragraph.

One big win of this approach is the **possibility to scale web and worker processes independently**, on the same or on different machines.

For instance if your application is an high-traffic one, with little side-work generated, you can deploy more web processes than worker ones, while if there are few network

requests that generates a lot of jobs for the worker, you can redistribute your resources accordingly.

Kue

In order to make web and worker processes talk to each other, a **queue** is a flexible approach that lets you not to worry about inter-process communication.

Kue is a common queue library for Node.js, it is based on Redis and lets you put in communication processes that are spawned on the same or on different machines in the exact same way.

Any kind of process can create a job and put it in the queue, then worker processes are configured to pick these jobs and execute them. A lot of options can be provided for each job, like priority, TTL, delay etc.

The more worker processes you spawn, the more parallel throughput you have to execute these jobs.

Cron

It is common for an application to have some tasks that need to be **performed periodically**. Usually this kind of operations are managed through **cron jobs** at the OS level, where a single script is invoked from the outside of your application.

This approach introduces the need of extra work when deploying your application on a new machine, that makes the process uncomfortable if you want to automate the deploy.

A more comfortable way to achieve the same result is to use the **cron module available on NPM**. It lets you to define cron jobs inside the Node.js code, making it independent from the OS configuration.

According to the web/worker pattern described above, the worker process can create the cron, which invokes a function putting a new job on the queue periodically.

Using the queue makes it more clean and can take advantage of all the features offered by kue like priority, retries, etc.

The problems come out when you have more than one worker process, because the cron function would wake app on every process at the same time, putting into the

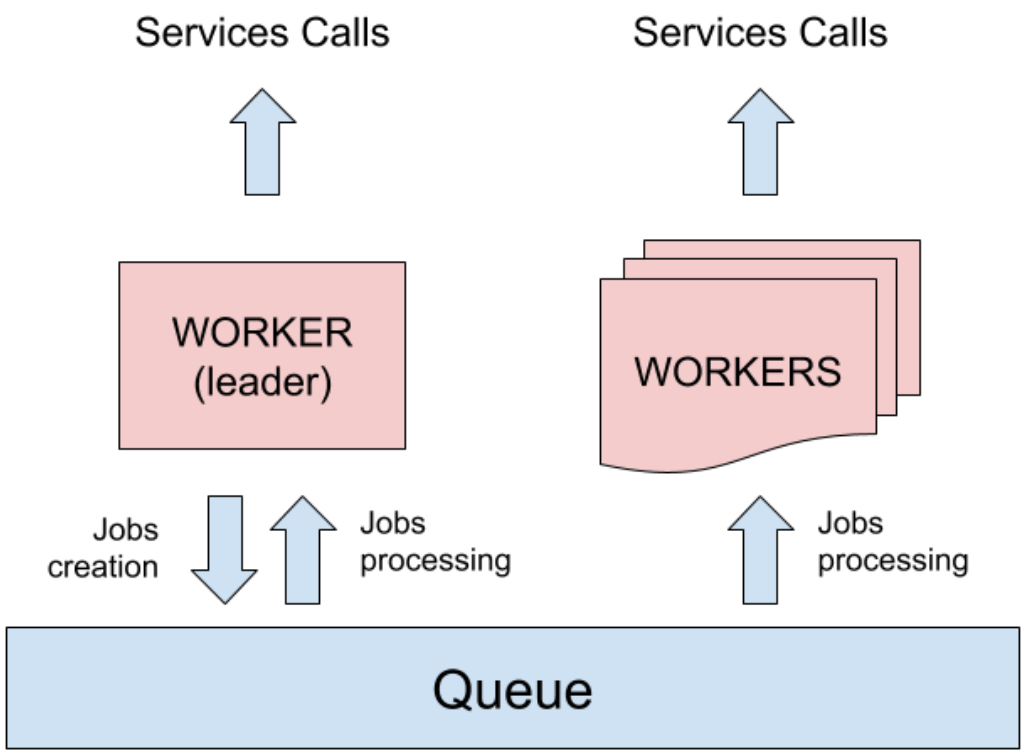
queue duplicates of the same job that would be executed many times.

In order to solve this problem it is necessary to **identify a single worker process that will perform the cron operations**.

Leader election and cron-cluster

This kind of problem is known as "**leader election**", and for this specific scenario there is an NPM package that does the trick for us, called cron-cluster.

It expose the same API that powers the cron module, but during setup it requires a **redis connection** used to communicate with the other processes and perform the leader election algorithm.



Using redis as a single source of truth, **all the process will agree about who will execute the cron**, and only one copy of the jobs will be put in the queue. After that, all the worker processes will be eligible to execute the jobs as usual.

Caching API calls

Server-side caching is a common way to **improve performance and reactivity** of your API calls, but it is a very wide topic with with a lot of possible implementations.

In a distributed environment like the one we described in this serie, using redis to store the cached values is probably the best approach in order to have all the nodes behaving equally.

The most difficult aspect to consider with caching is its invalidation. The quick-and-dirty solution considers only time, so the values in cache are flushed after a fixed TTL, with the downside of having to wait the next flush in order to see updates in the responses.

If you have more time to spend, it would be a better idea to implement the invalidation at the application level, manually flushing the records on the redis cache when the values change on DB.

Conclusions

In this serie of articles we covered some general topics about scaling and performance. The suggestions provided can be taken as a guideline that need to be customized on the particular need of your project.

Stay tuned for other articles about vertical topics on Node.js and DevOps!

. . .

If you like this article, clap as much as you want!

JavaScript Scaling Developer DevOps