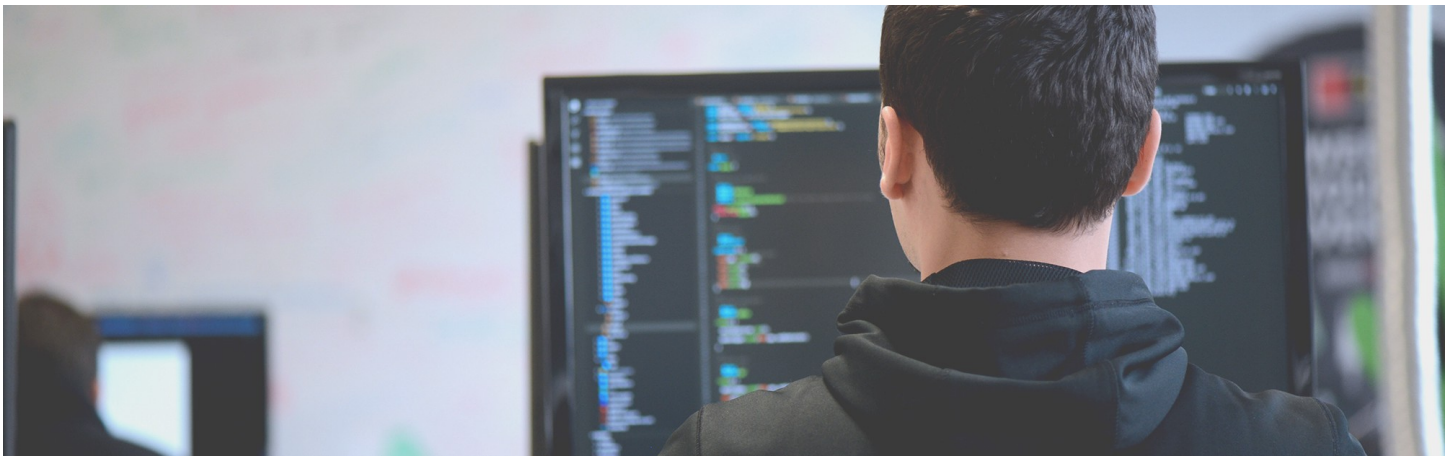
Good practices for high-performance and scalable Node.js applications [Part 2/3]

virgafox
Feb 27, 2018 · 5 min read



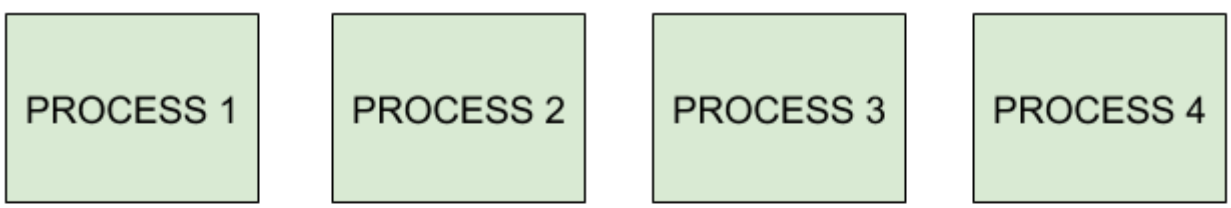
Chapter 2 — How to make your Node.js app ready to scale

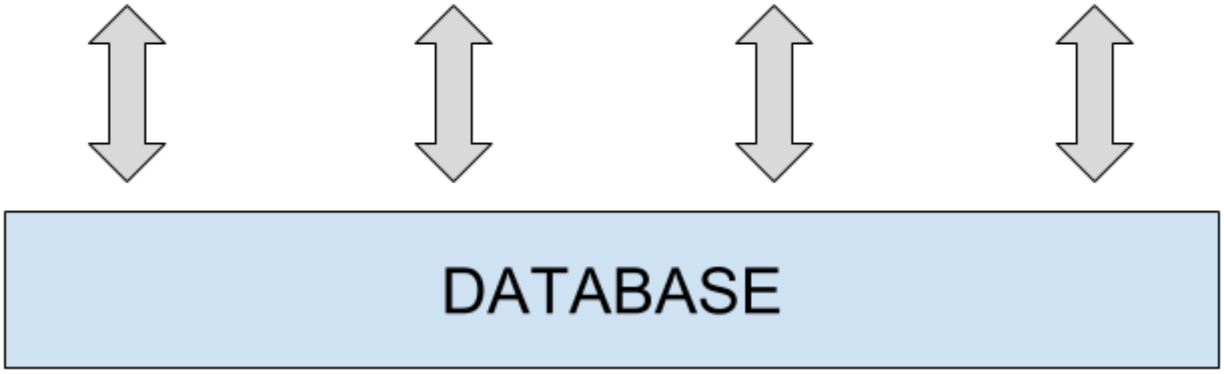
In the last article we saw how to horizontally scale a Node.js application, without worrying about the code. This time we'll talk about some aspects you must consider in order to prevent undesired behaviours when you scale up your process.

Decouple application instances from DB

The first hint isn't about code, but about your **infrastructure**.

If you want your application to be able to scale across different hosts, you must deploy your database on independent machines, so you can freely duplicate your application machine as you wish.





Deploying application and database on the same machine can be cheap and used for development purpose, but it is absolutely not recommended for production environments, where application and database must be able to scale independently. The same applies to in-memory databases like Redis.

Be stateless

If you spawn multiple instances of your application, **each process will have its own memory space**. This means that even if you are running on a single machine, when you store some value in a global variable, or more commonly a session in memory, you won't find it there if the balancer redirects you to another process during the next request.

This applies both to session data and internal values like any sort of app-wide settings.

For settings or configurations that can change during run-time, the solution is to store them on the external database (on storage or in-memory) in order to make it accessible by all processes.

Stateless authentication with JWT

Authentication is one of the first topics to consider when developing a stateless application. If you store sessions in memory, they will be scoped to that single process.

In order to make things work, you should configure your network load balancer to redirect the same user always to the same machine, and your local one to redirect the same user always to the same process (sticky sessions).

A trivial solution to this problem is to set the sessions' storage strategy to any form of persistence, for example storing them on DB instead of RAM. But if your application checks session data on each request, there will be disk I/O on every API call, that is definitely not good from the performance point of view.

A better and fast solution (if your authentication framework supports it) is to store sessions on an in-memory DB like Redis. A Redis instance is usually external to application instances like the DB one, but working in memory makes it much faster. Anyway storing sessions in RAM makes you need more memory when your concurrent sessions number grows.

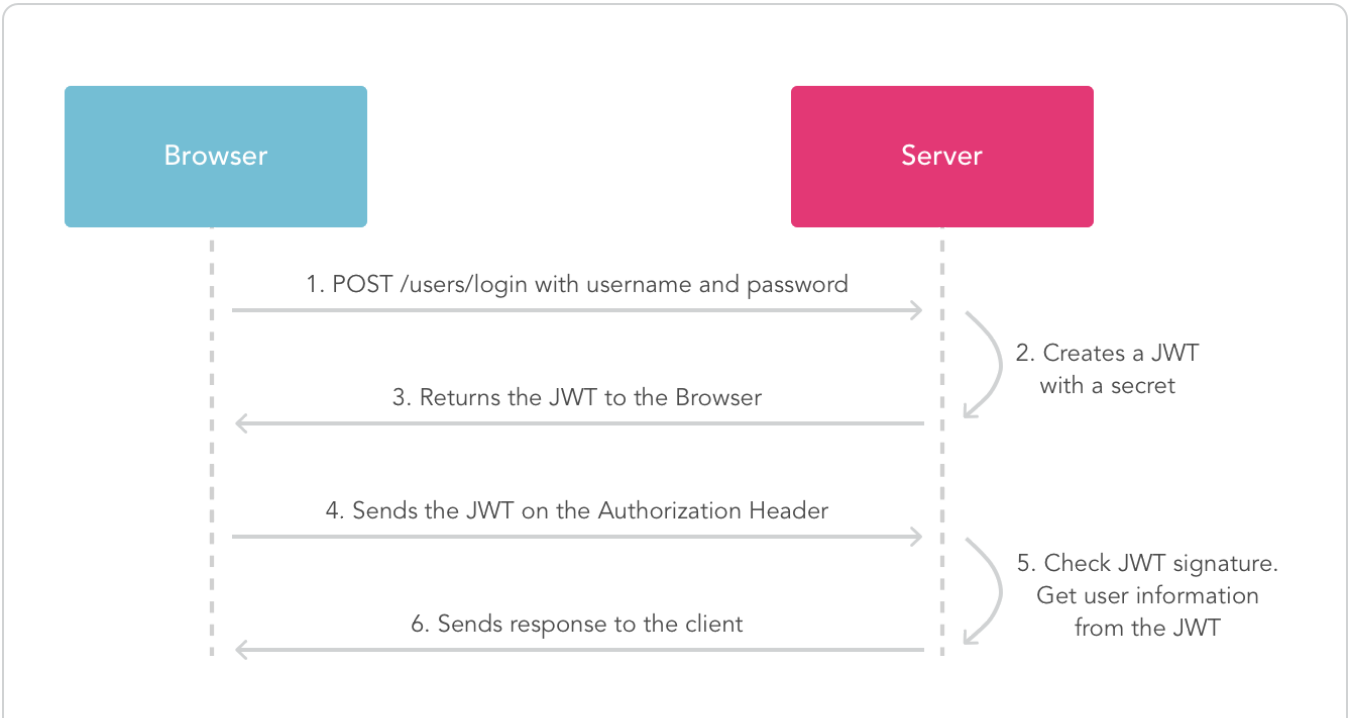
If you want to embrace a more efficient approach to stateless authentication, you can take a look a **JSON Web Tokens**.

The idea behind JWT is quite simple: When a user logs in, the server generates a token that is essentially a base64 encode 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 userID and its associated ACL roles. The token is sent back to the client and used by it to authenticate every API request.

When the server process an incoming request, it takes the payload of the token and recreates the signature using its secret key. If the two signatures match, the payload can be considered valid and not altered, and the user can be identified.

It's important to remember that **JWT doesn't provide any form of encryption**. The payload is only encoded in base64 and it is sent in clear text, so if you need to hide the content you must use SSL.

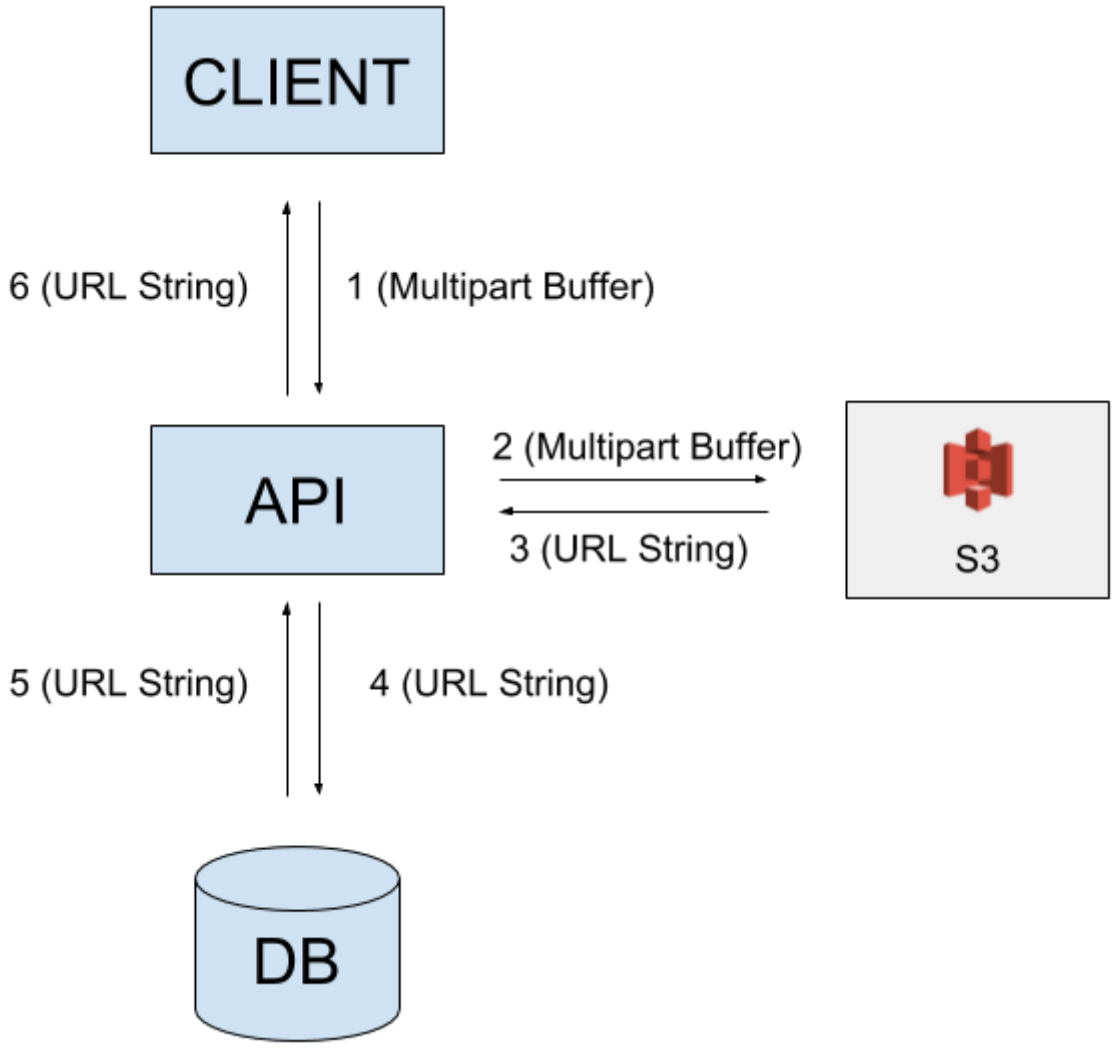
The following schema borrowed by jwt.io resumes the authentication process:



During the authentication process the server doesn't need to access session data stored somewhere, so each request can be processed by a different process or machine in a very efficient way. No data is saved in RAM, and you don't need to perform storage I/O, so this approach is really useful as you scale up.

Storage on S3

When you use multiple machines, you can't save user-generated assets directly on the filesystem, because those files would be accessible only to the processes local to that server. The solution is to **store all the content on an external service**, possibly on a dedicated one like Amazon S3, and save in your DB only the absolute URL that points to that resource.



Every process/machine will then have access to that resource in the same way.

Using the official AWS sdk for Node.js is quite easy and lets you integrate the service inside your application without particular effort. S3 is quite cheap and it is optimized for this purpose, making it a good choice also in the case your app isn't multi-process.

Properly configure WebSockets

If your application uses WebSockets for real-time interaction between clients or between client and server, you will need to **link your backend instance**s in order to correctly propagate broadcast messages, or messages between clients connected to different nodes.

The Socket.io library provides a special adapter for this purpose, called socket.io-redis, that lets you link your server instances using Redis pub-sub functionality.

In order to use a multi-node socket.io environment you will also need to force the protocol to "websockets", because long-polling needs sticky-sessions to work.

Next steps

In this short article we have seen some simple aspects to care about if you want to scale your Node.js app, that can be considered good practices also for single node environments.

In the next, and last, article of this small serie we'll see some additional patterns that can make your application perform better. You can find it here.

. . .

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

JavaScript Developer Nodejs Scaling Jwt