**1. Always Use Asynchronous Functions**Async functions are like the heart of JavaScript. So, the best we can do to optimize CPU usage is to write asynchronous functions to perform nonblocking I/O operations.

I/O operations include the processes which perform read and write data operations. It can be the database, cloud storage, or any local storage disk on which the I/O operations are performed.

Using asynchronous functions in an application that heavily uses I/O operations will improve it. This is because the CPU will be able to handle multiple requests simultaneously due to non-blocking I/O, while one of these requests is making an Input/Output operation.  
  
Here's an example:  
  
var fs = require('fs');

// Performing a blocking I/O

var file = fs.readFileSync('/etc/passwd');

console.log(file);

// Performing a non-blocking I/O

fs.readFile('/etc/passwd', function(err, file) {

if (err) return err;

console.log(file);

});

* We use the **fs**Node package to work with files.
* **readFileSync()**is synchronous and blocks execution until finished.
* **readFile()** is asynchronous and returns immediately while things function in the background.

## 2. Optimize Database Queries

Query optimization is an essential part of building optimized APIs in Node. Especially in larger applications, you'll need to query databases many times. So, a bad query can reduce the overall performance of the application.

Indexing is an approach to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique that is used to quickly locate and access the data in a database. Indexes are created using a few database columns.

Let's say we have a DB schema without indexing and the database contains 1 million records. A simple find query will go through a larger number of records to find the matching one compared to the schema with indexing.

* Query without indexing:

db.user.find({email: 'ofan@skyshi.com'}).explain("executionStats")

* Query with indexing:

db.getCollection("user").createIndex({ "email": 1 }, { "name": "email\_1", "unique": true })

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"ok" : 1

}

## ****3. Use Error Scripts with Logging****

The best way to monitor the proper functioning of your APIs is to keep track of their activity. This is where logging the data comes into play.

A common example of logging is printing out the logs to the console (using console.log()).

More efficient logging modules as compared to console.log are Morgan, Buyan, and Winston. Here, I’ll go with the example of Winston.

### How to log with Winston – features

* Provides 4 custom levels that we can use such as info, error, verbose, debug, silly, and warn.
* Supports querying the logs
* Simple profiling
* You can use multiple transports of the same type
* Catches and logs uncaughtException

You can set up Winston with the following command:

**4. Use HTTP/2 Instead of HTTP**

In addition to these techniques, we can also apply some other techniques like using HTTP/2 over HTTP, as it has the following advantages:

* Multiplexing
* Header compression
* Server push
* Binary format

It focuses on the performance and issues that the previous version of HTTP has. It makes web browsing faster and easier and consumes less bandwidth.

## ****5. Run Tasks in Parallel****

Use [async.js](https://caolan.github.io/async/v3/) to help you run tasks. Parallelizing tasks has a great impact on the performance of your API. It reduces latency and minimizes blocking operations.  
  
Parallel means running multiple things at the same time. However, when you run things in parallel, you don’t need to control the execution sequence of the program.

Here's a simple example using async parallel with an array:

const async = require("async");

// an example using an object instead of an array

async.parallel({

task1: function(callback) {

setTimeout(function() {

console.log('Task One');

callback(null, 1);

}, 200);

},

task2: function(callback) {

setTimeout(function() {

console.log('Task Two');

callback(null, 2);

}, 100);

}

}, function(err, results) {

console.log(results);

// results now equals to: {task2: 2, task1: 1}

});

in this example, we used [async.js](https://caolan.github.io/async/v3/) to execute the two tasks in asynchronous mode. Task 1 requires 200 ms to complete, but task 2 does not wait for its completion – it executes at its specified delay of 100ms.

Parallelizing tasks has a great impact on the performance of API. It reduces latency and minimizes blocking operations.

## ****6. Use Redis to Cache the App****

Redis is the advanced version of Memcached. It optimizes the APIs response time by storing and retrieving the data from the main memory of the server. It increases the performance of the database queries which also reduces access latency.

In the following code snippets, we have called the APIs without and with Redis, respectively, and compared the response time.