[Node.js](https://nodejs.org/) is becoming more and more popular as a server-side run-time environment, especially for high traffic websites, as [statistics show](http://w3techs.com/technologies/details/ws-nodejs/all/all). Also, the availability of several frameworks make it a good environment for rapid prototyping. Node.js has an event-driven architecture, leveraging a non-blocking I/O API that allows requests being processed asynchronously.

One of the important and often less highlighted features of Node.js is its scalability. In fact, this is the main reason why [some large companies with heavy traffic are integrating Node.js in their platform](https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node) (e.g., Microsoft, Yahoo, Uber, and Walmart) or even completely moving their server-side operations to Node.js (e.g., PayPal, eBay, and Groupon).

Each Node.js process runs in a single thread and by default it has a memory limit of 512MB on 32-bit systems and 1GB on 64-bit systems. Although [the memory limit can be bumped to ~1GB on 32-bit systems and ~1.7GB on 64-bit systems](https://github.com/joyent/node/wiki/FAQ#what-is-the-memory-limit-on-a-node-process), both memory and processing power can still become bottlenecks for various processes.

The elegant solution Node.js provides for scaling up the applications is to split a single process into multiple processes or *workers*, in Node.js terminology. This can be achieved through a cluster module. The cluster module allows you to create child processes (workers), which share all the server ports with the main Node process (master).

In this article you’ll see how to create a Node.js cluster for speeding up your applications.

**Node.js Cluster Module: what it is and how it works**

A cluster is a pool of similar workers running under a parent Node process. Workers are spawned using the fork() method of the child\_processes module. This means workers can share server handles and use IPC (Inter-process communication) to communicate with the parent Node process.

he master process is in charge of initiating workers and controlling them. You can create an arbitrary number of workers in your master process. Moreover, remember that by default incoming connections are distributed in a round-robin approach among workers (except in Windows). Actually there is another approach to distribute incoming connections, that I won’t discuss here, which hands the assignment over to the OS (default in Windows). [Node.js documentation](https://nodejs.org/api/cluster.html#cluster_cluster_schedulingpolicy) suggests using the default round-robin style as the scheduling policy.

Although using a cluster module sounds complex in theory, it is very straightforward to implement. To start using it, you have to include it in your Node.js application:

var cluster = require('cluster);

A cluster module executes the same Node.js process multiple times. Therefore, the first thing you need to do is to identify what portion of the code is for the master process and what portion is for the workers. The cluster module allows you to identify the master process as follows:

if(cluster.isMaster) { ... }

The master process is the process you initiate, which in turn initialize the workers. To start a worker process inside a master process, we’ll use the fork() method:

cluster.fork();

This method returns a worker object that contains some methods and properties about the forked worker. We’ll see some examples in the following section.

A cluster module contains several events. Two common events related to the moments of start and termination of workers are the online and the exit events. online is emitted when the worker is forked and sends the online message. exit is emitted when a worker process dies. Later, we’ll see how we can use these two events to control the lifetime of the workers.

Let’s now put together everything we’ve seen so far and show a complete working example.

**Examples**

This section features two examples. The first one is a simple application showing how a cluster module is used in a Node.js application. The second one is an Express server taking advantage of Node.js cluster module, which is part of a production code I generally use in large-scale projects. [Both the examples can be downloaded from GitHub](https://github.com/sitepoint-editors/node-cluster-tutorial).

**How a Cluster Module is Used in a Node.js App**

In [this first example](https://github.com/sitepoint-editors/node-cluster-tutorial/blob/master/simple.js), we set up a simple server that responds to all incoming requests with a message containing the worker process ID that processed the request. The master process forks four workers. In each of them, we start listening the port 8000 for incoming requests.

The code that implements what I’ve just described, is shown below:

var cluster = require('cluster');

var http = require('http');

var numCPUs = 4;

if (cluster.isMaster) {

for (var i = 0; i < numCPUs; i++) {

cluster.fork();

}

} else {

http.createServer(function(req, res) {

res.writeHead(200);

res.end('process ' + process.pid + ' says hello!');

}).listen(8000);

}

You can test this server on your machine by starting it (run the command node simple.js) and accessing the URL <http://127.0.0.1:8000/>. When requests are received, they are distributed one at a time to each worker. If a worker is available, it immediately starts processing the request; otherwise it’ll be added to a queue.

There are a few points that are not very efficient in the above example. For instance, imagine if a worker dies for some reason. In this case, you lose one of your workers and if the same happens again, you will end up with a master process with no workers to handle incoming requests. Another issue is related to the number of workers. There are different number of cores/threads in the systems that you deploy your application to. In the mentioned example, to use all of the system’s resources, you have to manually check the specifications of each deployment server, find how many threads there are available, and update it in your code. In the next example, we’ll see how to make the code more efficient through an Express server.

**How to Develop a Highly Scalable Express Server**

[Express](http://expressjs.com/) is one the most popular web application frameworks for Node.js (if not the most popular). On SitePoint we have covered it a few times. If you’re interested in knowing more about it, I suggest you to read the articles [Creating RESTful APIs with Express 4](http://www.sitepoint.com/creating-restful-apis-express-4/) and [Build a Node.js-powered Chatroom Web App: Express and Azure](http://www.sitepoint.com/build-node-js-powered-chatroom-web-app-expess-azure/).

This [second example](https://github.com/sitepoint-editors/node-cluster-tutorial/blob/master/express.js) shows how we can develop a highly scalable Express server. It also demonstrates how to migrate a single process server to take advantage of a cluster module with few lines of code.

var cluster = require('cluster');

if(cluster.isMaster) {

var numWorkers = require('os').cpus().length;

console.log('Master cluster setting up ' + numWorkers + ' workers...');

for(var i = 0; i < numWorkers; i++) {

cluster.fork();

}

cluster.on('online', function(worker) {

console.log('Worker ' + worker.process.pid + ' is online');

});

cluster.on('exit', function(worker, code, signal) {

console.log('Worker ' + worker.process.pid + ' died with code: ' + code + ', and signal: ' + signal);

console.log('Starting a new worker');

cluster.fork();

});

} else {

var app = require('express')();

app.all('/\*', function(req, res) {res.send('process ' + process.pid + ' says hello!').end();})

var server = app.listen(8000, function() {

console.log('Process ' + process.pid + ' is listening to all incoming requests');

});

}

The first addition to this example is getting the number of the CPU cores using the Node.js os module. The os module contains a cpus()function, which returns an array of CPU cores. Using this approach, we determine the number of the workers to fork dynamically, based on the server specifications to maximize the utilization.

A second and more important addition is handling a worker’s death. When a worker dies, the cluster module emits an exit event. It can be handled by listening for the event and executing a callback function when it’s emitted. You can do that by writing a statement like cluster.on('exit', callback);. In the callback, we fork a new worker in order to maintain the intended number of workers. This allows us to keep the application running, even if there are some unhandled exceptions.

In this example, I also set a listener for an online event, which is emitted whenever a worker is forked and ready to receive incoming requests. This can be used for logging or other operations.

**Performance Comparison**

There are several tools to benchmark APIs, but here I use Apache Benchmark tool to analyze how using a cluster module can affect the performance of your application.

To set up the test, [I developed an Express server](https://github.com/sitepoint-editors/node-cluster-tutorial/blob/master/express.js) that has one route and one callback for the route. In the callback, a dummy operation is performed and then a short message is returned. There are two versions of the server: one with no workers, in which everything happens in the master process, and the other with 8 workers (as my machine has 8 cores). The table below shows how incorporating a cluster module can increase the number of processed requests per second.

| **Concurrent Connections** | **1** | **2** | **4** | **8** | **16** |
| --- | --- | --- | --- | --- | --- |
| **Single Process** | 654 | 711 | 783 | 776 | 754 |
| **8 Workers** | 594 | 1198 | 2110 | 3010 | 3024 |

(Requests processed per second)

**Advanced Operations**

While using cluster modules is relatively straightforward, there are other operations you can perform using workers. For instance, you can achieve (almost!) zero down-time in your application using cluster modules. We’ll see how to perform some of these operations in a while.

**Communication Between Master and Workers**

Occasionally you may need to send messages from the master to a worker to assign a task or perform other operations. In return, workers may need to inform the master that the task is completed. To listen for messages, an event listener for the message event should be set up in both master and workers:

worker.on('message', function(message) {  
    console.log(message);  
});

The worker object is the reference returned by the fork() method. To listen for messages from the master in a worker:

process.on('message', function(message) {  
    console.log(message);  
});

Messages can be strings or JSON objects. To send a message from the master to a specific worker, you can write a code like the on reported below:

worker.send('hello from the master');

Similarly, to send a message from a worker to the master you can write:

process.send('hello from worker with id: ' + process.pid);

In Node.js, messages are generic and do not have a specific type. Therefore, it is a good practice to send messages as JSON objects with some information about the message type, sender, and the content itself. For example:

worker.send({  
    type: 'task 1',  
    from: 'master',  
    data: {  
        // the data that you want to transfer  
    }  
});

An important point to note here is that message event callbacks are handled asynchronously. There isn’t a defined order of execution. [You can find a complete example of communication between the master and workers on GitHub](https://github.com/sitepoint-editors/node-cluster-tutorial/blob/master/communication.js).

**Zero Down-time**

One important result that can be achieved using workers is (almost) zero down-time servers. Within the master process, you can terminate and restart the workers one at a time, after you make changes to your application. This allows you to have older version running, while loading the new one.

To be able to restart your application while running, you have to keep two points in mind. Firstly, the master process runs the whole time, and only workers are terminated and restarted. Therefore, it’s important to keep your master process short and only in charge of managing workers.

Secondly, you need to notify the master process somehow that it needs to restart workers. There are several methods for doing this, including a user input or watching the files for changes. The latter is more efficient, but you need to identify files to watch in the master process.

My suggestion for restarting your workers is to try to shut them down safely first; then, if they did not safely terminate, forcing to kill them. You can do the former by sending a shutdown message to the worker as follows:

workers[wid].send({type: 'shutdown', from: 'master'});

And start the safe shutdown in the worker message event handler:

process.on('message', function(message) {  
    if(message.type === 'shutdown') {  
        process.exit(0);  
    }  
});

To do this for all the workers, you can use the workers property of the cluster module that keeps a reference to all the running workers. We can also wrap all the tasks in a function in the master process, which can be called whenever we want to restart all the workers.

function restartWorkers() {

var wid, workerIds = [];

for(wid in cluster.workers) {

workerIds.push(wid);

}

workerIds.forEach(function(wid) {

cluster.workers[wid].send({

text: 'shutdown',

from: 'master'

});

setTimeout(function() {

if(cluster.workers[wid]) {

cluster.workers[wid].kill('SIGKILL');

}

}, 5000);

});

};

We can get the ID of all the running workers from the workers object in the cluster module. This object keeps a reference to all the running workers and is dynamically updated when workers are terminated and restarted. First we store the ID of all the running workers in a workerIds array. This way, we avoid restarting newly forked workers.

Then, we request a safe shutdown from each worker. If after 5 seconds the worker is still running and it still exists in the workersobject, we then call the kill function on the worker to force it shutdown. [You can find a practical example on GitHub](https://github.com/sitepoint-editors/node-cluster-tutorial/blob/master/zero_downtime.js).

**Conclusions**

Node.js applications can be parallelized using cluster modules in order to use the system more efficiently. Running multiple processes at the same time can be done using few lines of code and this makes the migration relatively easy, as Node.js handles the hard part.

As I showed in the performance comparison, there is potential for noticeable improvement in the application performance by utilizing system resources in a more efficient way. In addition to performance, you can increase your application’s reliability and uptime by restarting workers while your application is running.

That being said, you need to be careful when considering the use of a cluster module in your application. The main recommended use for cluster modules is for web servers. In other cases, you need to study carefully how to distribute tasks between workers, and how to efficiently communicate progress between the workers and the master. Even for web servers, make sure a single Node.js process is a bottleneck (memory or CPU), before making any changes to your application, as you might introduce bugs with your change.

One last thing, Node.js website has a great [documentation for cluster module](https://nodejs.org/api/cluster.html). So, be sure to check it out!