**Difference between cluster and child\_process modules?**

*cluser.fork is implemented on top of child\_process.fork. The extra stuff that cluster.fork brings is that, it will enable you to listen on a shared port. If you don't want it, just use child\_process.fork. So yeah, use cluster for web servers and child\_process for workers.*

**What does emitter do and what is dispatcher?**

*The scalability referred to here is more about scaling the codebase than scaling in terms of how fast the software is. Data in flux systems is easy to trace because every store is registered to every action, and the actions define every app-wide event that can happen in the system. Each store can determine how it needs to update itself in response to each action, without the programmer needing to decide which stores to wire up to which actions, and in most cases, you can change or read the code for a store without needing to worrying about how it affects any other store.*

*Since v6.0.X Node.js has included the cluster module straight out of the box, which makes it easy to set up multiple node workers that can listen on a single port. Note that this is NOT the same as the older learnboost "cluster" module available through npm.*

*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);*

[***https://stackoverflow.com/questions/2387724/node-js-on-multi-core-machines***](https://stackoverflow.com/questions/2387724/node-js-on-multi-core-machines)

**How to deploy Node.js in cloud for high availability using multi-core, reverse-proxy, and SSL**

*It's been several months since I asked this question and not a lot of answer flow. Both Samyak Bhuta and nponeccop had good suggestions, but I wanted to discuss the answers I've found to my questions.*

*Here is what I've settled on at this point for a production system, but further improvements are always being made. I hope it helps anyone in a similar scenario.*

*Use Cluster to spawn as many child processes as you desire to handle incoming requests on multi-core virtual or physical machines. This binds to a single port and makes maintenance easier. My rule of thumb is n - 1 Cluster workers. You don't need Forever on this, as Cluster respawns worker processes that die. To have resiliency even at the Cluster parent level, ensure that you use an Upstart script (or equivalent) to daemonize the Node.js application, and use Monit (or equivalent) to watch the PID of the Cluster parent and respawn it if it dies. You can try using the respawn feature of Upstart, but I prefer having Monit watching things, so rather than split responsibilities, I find it's best to let Monit handle the respawn as well.*

*Use 1 nginx per app server running on port 80, simply reverse proxying to your Cluster on whatever port you bound to in (1). node-http-proxy can be used, but nginx is more mature, more featureful, and faster at serving static files. Run nginx lean (don't log, don't gzip tiny files) to minimize it's overhead.*

*Have minimum 2x servers as described above in a minimum of 2 availability zones, and if in AWS, use an ELB that terminates HTTPS/SSL on port 443 and communicates on HTTP port 80 to the node.js app servers. ELBs are simple and, if you desire, make it somewhat easier to auto-scale. You could run multiple nginx either sharing an IP or round-robin balanced themselves by your DNS provider, but I found this overkill for now. At that point, you'd remove the nginx instance on each app server.*

*I have not needed WebSockets so nginx continues to be suitable and I'll revisit this issue when WebSockets come into the picture.*

*Feedback is welcome.*

***Writing files in Node.js***

*There are a lot of details in the filesystem API. The most common way (as far as I know) is:*

*var fs = require('fs');*

*fs.writeFile("/tmp/test", "Hey there!", function(err) {*

*if(err) {*

*return console.log(err);*

*}*

*console.log("The file was saved!");*

*});*

**In Node.js, I want to read a file, and then console.log() each line of the file separated by \n. How can I do that?**

*var fs=require('fs');*

*fs.readFile('/path/to/file','utf8', function (err, data) {*

*if (err) throw err;*

*var arr=data.split('\n');*

*arr.forEach(function(v){*

*console.log(v);*

*});*

*});*

**How do I include a JavaScript file in another JavaScript file?**

*The old versions of JavaScript had no import, include, or require, so many different approaches to this problem have been developed.*

*But since 2015 (ES6), JavaScript has had the ES6 modules standard to import modules in Node.js, which is also supported by most modern browsers.*

*For compatibility with older browsers, build and/or transpilation tools can be used.*

*ES6 Modules*

*ECMAScript (ES6) modules have been supported in Node.js since v8.5, with the --experimental-modules flag. All files involved must have the .mjs extension.*

*// module.mjs*

*export function hello() {*

*return "Hello";*

*}*

*// main.mjs*

*import { hello } from 'module'; // or '.*

**var functionName = function() {} vs function functionName() {} different**

*The difference is that functionOne is a function expression and so only defined when that line is reached, whereas functionTwo is a function declaration and is defined as soon as its surrounding function or script is executed (due to hoisting).*

*For example, a function expression:*

*// TypeError: functionOne is not a function*

*functionOne();*

*var functionOne = function() {*

*console.log("Hello!");*

*};*

[*https://stackoverflow.com/questions/336859/var-functionname-function-vs-function-functionname?rq=1*](https://stackoverflow.com/questions/336859/var-functionname-function-vs-function-functionname?rq=1)

***What's the difference between using “let” and “var” to declare a variable in JavaScript?***

*The difference is scoping. var is scoped to the nearest function block and let is scoped to the nearest enclosing block, which can be smaller than a function block. Both are global if outside any block.*

*Also, variables declared with let are not accessible before they are declared in their enclosing block. As seen in the demo, this will throw a ReferenceError exception.*

*Demo:*

*Show code snippet*

*Global:*

*They are very similar when used like this outside a function block.*

*let me = 'go'; // globally scoped*

*var i = 'able'; // globally scoped*

*However, global variables defined with let will not be added as properties on the global window object like those defined with var.*

*console.log(window.me); // undefined*

*console.log(window.i); // 'able'*

*Function:*

*They are identical when used like this in a function block.*

*function ingWithinEstablishedParameters() {*

*let terOfRecommendation = 'awesome worker!'; //function block scoped*

*var sityCheerleading = 'go!'; //function block scoped*

*}*

*Block:*

*Here is the difference. let is only visible in the for() loop and var is visible to the whole function.*

*function allyIlliterate() {*

*//tuce is \*not\* visible out here*

*for( let tuce = 0; tuce < 5; tuce++ ) {*

*//tuce is only visible in here (and in the for() parentheses)*

*//and there is a separate tuce variable for each iteration of the loop*

*}*

*//tuce is \*not\* visible out here*

*}*

*function byE40() {*

*//nish \*is\* visible out here*

*for( var nish = 0; nish < 5; nish++ ) {*

*//nish is visible to the whole function*

*}*

*//nish \*is\* visible out here*

*}*

*Redeclaration:*

*Assuming strict mode, var will let you re-declare the same variable in the same scope. On the other hand, let will not:*

*'use strict';*

*let me = 'foo';*

*let me = 'bar'; // SyntaxError: Identifier 'me' has already been declared*

*'use strict';*

*var me = 'foo';*

*var me = 'bar'; // No problem, `me` is replaced.*

***How do I debug Node.js applications?***

*node-inspector could save the day! Use it from any browser supporting WebSocket. Breakpoints, profiler, livecoding, etc... It is really awesome.*

*Install it with:*

*npm install -g node-inspector*

*Then run:*

*node-debug app.js*

**Node.js Control Flow and Callbacks**

*You should be using promises. Bluebird is a great promise library. Faster than native and comes with great features. With promises you can chain together functions, and know that one will not be called until the previous function resolves. No need to set timeouts or delays. Although you can if you'd like. Below is example of a delay. Function B wont run until 6 seconds after A finishes. If you remove .delay(ms) B will run immediately after A finishes.*

*var Promise = require("bluebird");*

*console.time('tracked');*

*console.time('first');*

*function a (){*

*console.log('hello');*

*console.timeEnd('first');*

*return Promise.resolve();*

*}*

*function b (){*

*console.log('world');*

*console.timeEnd('tracked');*

*}*

*a().delay(6000)*

*.then(b)*

*.catch(Promise.TimeoutError, function(e) {*

*console.log('Something messed up yo', e);*

*});*