# Q-. What Is A Child\_process Module In Node.Js?

Answer.

Node.js supports the creation of child processes to help in parallel processing along with the event-driven model.

The child\_process module enables us **to access Operating System** functionalities by running any system command inside a, well, child process.

We can control that child process input stream, and listen to its output stream. We can also control the arguments to be passed to the underlying OS command, and we can do whatever we want with that command’s output. We can, for example, pipe the output of one command as the input to another (just like we do in Linux) as all inputs and outputs of these commands can be presented to us using [Node.js streams](https://medium.freecodecamp.com/node-js-streams-everything-you-need-to-know-c9141306be93).

The Child processes always have three streams <child.stdin>, child.stdout, and child.stderr. The <stdio> stream of the parent process shares the streams of the child process.

Node.js provides a <child\_process> There are four different ways to create a child process in Node: spawn(), fork(), exec(), and execFile().

* **exec** – <child\_process.exec> method runs a command in a shell/console and buffers the output.
* **spawn** – <child\_process.spawn> launches a new process with a given command.
* **fork** – <child\_process.fork> is a special case of the spawn() method to create child processes.

# Clustering

What is clustering ? Clustering in Node.js allows you to create separate processes which can share same server port. For example, if we run one HTTP server on Port 3000, it is one Server running on Single thread on a single core of the processor. But I want to take advantage of all core available in my machine. So I will cluster my application and run them on all cores. So if I run one server on Port 3000 by having 4 core of processor then actually I am running 4 servers all are listening to Port 3000. So if one server goes down then other is there to take the place of it, also in peak load of traffic, Node will automatically allocate the worker to particular process so basically it does internal load balancing very efficiently.

A single instance of **Node**.**js** runs in a single thread. To take advantage of multi-core systems, the user will sometimes want to launch a **cluster** of **Node**.**js** processes to handle the load. The **cluster** module allows easy creation of child processes that all share server ports.

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);

}

Another example

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');

});

}

Here important addition is handling a worker’s death. When a worker dies, the cluster module emits an exit event. 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.

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.

### 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 messageevent 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

*https://www.sitepoint.com/how-to-create-a-node-js-cluster-for-speeding-up-your-apps/*

# Micro-services

Read haapi.js

A microservice architecture means that your app is made up of lots of smaller, independent applications capable of running in their own memory space and scaling independently from each other across potentially many separate machines

A microservice is a single self-contained unit which, together with many others, makes up a large application. By splitting your app into small units every part of it is independently deployable and scalable, can be written by different teams and in different programming languages and can be tested individually.

The application starts faster, which makes developers more productive, and speeds up deployments. Each service can be deployed independently of other services—easier to deploy new versions of services frequently

# Arrow Function

Its like beta function

An **arrow function expression** has a shorter syntax than a [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function) and does not have its own [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), or [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target). These function expressions are best suited for non-method functions, and they **cannot be used as constructors.**

we avoid having to type the function keyword, return keyword (it’s implicit in arrow functions), and curly brackets

 Arrow functions are anonymous and change the way this binds in functions.

The goal of Arrow Functions is to address and resolve several common pain points of traditional Function Expression:

* Lexical this binding;
* Shorter syntactical form (() => {} vs. function () {})

An arrow function does not have its own this; the this value of the enclosing execution context is used. Thus, in the following code, the this within the function that is passed to setInterval has the same value as this in the enclosing function:

function Person(){

this.age = 0;

setInterval(() => {

this.age++; // |this| properly refers to the person object

}, 1000);

}

var p = new Person();

**Benefit #1: Shorter Syntax**

function funcName(params) {  
 return params + 2;  
 }

funcName(2);  
// 4

This above code indicates one of the two reasons for creating arrow functions: **shorter syntax.**The exact same functions can be expressed as an arrow function with only one line of code:

var funcName = (params) => params + 2

funcName(2);  
// 4

Pretty cool. This example is obviously an extreme simplification, but hopefully illustrates my point. Lets take a look at the syntax of arrow functions a little more in-depth:

**(**parameters**) => {** statements **}**

If we have no parameters, we express an arrow function like this:

**() => {** statements **}**

When you only have one parameter, the opening parenthesis are optional:

parameters **=> {** statements **}**

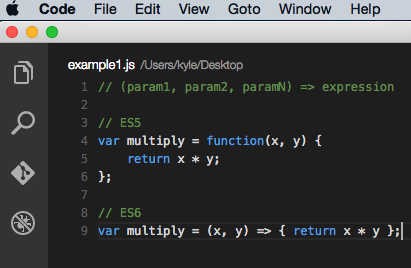
Finally, if you are returning an expression, you remove the brackets:

parameters **=>** expression

// is equivalent to:

function (parameters){  
 return expression;  
}

var double = num => num \* 2

s

### Basic Syntax with One Parameter

### No Parameters

### Benefit #2: No binding of this

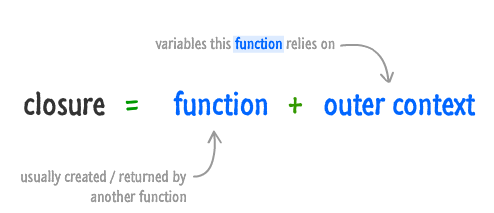
Before we move on, you should have a good understanding of the keyword thisand how it works. If you want to learn, or need a refresher, read [my post](https://codeburst.io/javascript-the-keyword-this-for-beginners-fb5238d99f85) on the subject before continuing.

Unlike a regular function, an arrow function does not bind this. Instead,this is bound lexically (i.e. this keeps its meaning from its original context).

# closure:

Closure means a function inside a function which can access variables declared outside it.

Basically it is accessing a variable which is out of scope.



**closure** is an inner function which gets returned by outer function and this inner function is holding outer context .. like variables declared in outer function

function outer() {

    alert("Good Bye!");

    function inner() {

        alert("Hello!");

    }

    return inner();

}

**Var callFun = outer(); // how to call**

**callFun();**

**Why clouser**: With a function closure you can store data in a separate scope, and share it only where necessary.

**The most important thing closures do is allow functions to keep on working even if their environment drastically changes or disappears.** Any variables that were in scope when the function was created are enclosed and protected to ensure the function still works. This behavior is essential for a very dynamic language like JavaScript where you often create, modify, and destroy things on the fly. Yay!

Inner functions store their outer function’s variables by reference, not by value.

function cityLocation() {

var city = "Paris";

return {

get: function() { console.log(city); },

set: function(newCity) { city = newCity; }

};

}

var myLocation = cityLocation();

myLocation.get(); // output: Paris

myLocation.set('Sydney');

myLocation.get(); // output: Sydney

# Lexical Scoping

**The scope of variables is defined by their position in source code**.

First off, JavaScript has *lexical scoping* with *function scope*. In other words, even though JavaScript looks like it should have block scope because it uses curly braces { }, a new scope is created only when you create a new function.

|  |
| --- |
| **var** outerFunction  = **function**(){    **if**(**true**){  **var** x = 5;  *//console.log(y); //line 1, ReferenceError: y not defined*  }    **var** nestedFunction = **function**() {    **if**(**true**){  **var** y = 7;  console.log(x); *//line 2, x will still be known prints 5*  }    **if**(**true**){  console.log(y); *//line 3, prints 7*  }  }  **return** nestedFunction;  }    **var** myFunction = outerFunction();  myFunction(); |

In this example, the variable *x* is available everywhere inside of outerFunction(). Also, the variable *y* is available everywhere within the nestedFunction(), but neither are available outside of the function where they were defined. The reason for this can be explained by lexical scoping. **The scope of variables is defined by their position in source code**. In order to resolve variables, JavaScript starts at the innermost scope and searches outwards until it finds the variable it was looking for. Lexical scoping is nice, because we can easily figure out what the value of a variable will be by looking at the code; whereas in dynamic scoping, the meaning of a variable can change at runtime, making it more difficult.

## Closures

The fact that we can access the variable *x* might still be confusing, because, normally, a local variable inside a function is gone after a function finishes executing. We called outerFunction() and assigned its result, nestedFunction(), into myFunction(). How does the variable *x* still exist if outerFunction() has already returned?

Merely accessing a variable outside of the immediate scope (no return statement is necessary) will create something called a *closure*

# Web Socket’s –

Web socket is a way to communicate between client (browser) and server

It has bidirectional data flow – from client to server and also from server to client

Web sockets are always open so that it allows real time data flow in application

# variable hoisting in javascript

## **JavaScript Declarations are Hoisted**

In JavaScript all the declarations are hoisted (moved) up to the top **of their scope**. This applies to both function and variable declarations.

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

x=5;

y=8;

console.log("value of + y is "+(x+y)); // output is 13

var x;

var y;

## **JavaScript Initializations are Not Hoisted**

JavaScript only hoists declarations, not initializations.

var x=5;

console.log("value of + y is "+(x+y)); // value of y is not available

// **output is NaN**

var y = 8;

## **Declare Your Variables At the Top !**

Hoisting is an unknown or overlooked behavior of JavaScript.

If a developer doesn't understand hoisting, programs may contain bugs (errors).

To avoid bugs, always declare all variables at the beginning of every scope.

JavaScript in strict mode does not allow variables to be used if they are not declared.  
Study **"use strict"**

# Difference Bet Put and POST

In PUT method, no matter how many times we send the same request, the results will always be the same. On the other hand, the POST method, we send the same POST request multiple times, we will receive various results means a new subordinate will be created each time

# Q express-generator

express-generator is used to quickly create an application skeleton.

$ npm install express-generator -g

The generated app has the following directory structure:

.

├── app.js

├── bin

│ └── www

├── package.json

├── public

│ ├── images

│ ├── javascripts

│ └── stylesheets

│ └── style.css

├── routes

│ ├── index.js

│ └── users.js

└── views

├── error.pug

├── index.pug

└── layout.pug

7 directories, 9 files

# Q Difference between let and var

3 differences

1. Var is earlier and let is latest in ES 6
2. Var has function scope and let has block scope
3. Variables declared with var will get hoisted

Example :

var

function nodeSimplified(){

var a =10;

console.log(a); // output 10

if(true){

var a=20;

console.log(a); // output 20

}

console.log(a); // output 20

}

let

function nodeSimplified(){

let a =10;

console.log(a); // output 10

if(true){

let a=20;

console.log(a); // output 20 ///block scope

}

console.log(a); // output 10

}

It is almost the same behavior we see in most language.

function nodeSimplified(){

let a =10;

let a =20; //throws syntax error

console.log(a);

}

**Error Message: Uncaught SyntaxError: Identifier 'a' has already been declared.**

However, with var, it works fine.

function nodeSimplified(){

var a =10;

var a =20;

console.log(a); //output 20

}

Const

function nodeSimplified(){

const MY\_VARIABLE =10;

console.log(MY\_VARIABLE); //output 10

MY\_VARIABLE =20; //throws type error

console.log(MY\_VARIABLE);

}

**Error Message : Uncaught TypeError: Assignment to constant variable.**

* Variable Hoisting

let will not hoist to the entire scope of the block they appear in. By contrast, var could hoist as below.

{

console.log(c); // undefined. Due to hoisting

**var** c = 2;

}

{

console.log(b); // ReferenceError: b is not defined

**let** b = 3;

}

**let**, unlike **var**, does not create a property on the global object. For example:

var x = 'global';

let y = 'global';

console.log(this.x); // "global"

console.log(this.y); // undefined

# Q Difference between let and const

The use of*let*and *const*is done as an alternative to the *var*function during the declaration of variables. Unlike *var*,*let*and *const*are block scoped. Another difference that *let*and *const*havewhen compared to *var*is the fact that while*var* gives a *undefined* error when accessed before declaring, *let*and *const*give a *ReferenceError* if they are accessed before declaration.

Case 1 const does not allow reassignment

const a =5;

a=9;

error: **TypeError**: Assignment to constant variable.

Case 2 : If a=5 and a=9 .. for both const keyword is used

const a =5;

const a =9;

error: **SyntaxError**: Identifier 'a' has already been declared

**Let works in CASE 1 ===it wont give error ..CASE 2 is same as const**

Case 1 const does not allow reassignment

let a =5;

a=9;

console.log(a) // 9

Case 2 : If a=5 and a=9 .. for both const keyword is used

let a =5;

let a =9;

error: **SyntaxError**: Identifier 'a' has already been declared

Also declaration of the variable with the same name will throw an error. This helps to fix mistakes where one loses a reference when a variable is reassigned.

# Q const are mutable

variables declared with const are mutable(can be chaned). If a variable is declared as a primitive like the following, const foo = 123, it is immutable(can not be changed) but if a variable is declared as an object(const bar = {}) it is mutable. This happens because the new variable always points to the same object but the properties on that object can change. So, the following code is perfectly acceptable in ES6 and will probably not behave as you expect:

const my\_object = {

p1: 'bar'

};

my\_object.p2 = 'totally mutable';

my\_object.p1 = 'why no breakage?';

console.log(my\_object);

output: { p1: 'why no breakage?', p2: 'totally mutable' }

**NOTE: the property p1 has been changed and p2 has been newly added.**

**Const object allows both.**

On the other hand, the following code will throw a syntax error as expected:

const primitive = 1234;

primitive = 5678; // Throws an error!

# Q Diff undefined and not defined

**Undefined –**

In case of var… accessing a variable before declaration …

Or before giving any value ..is UNDEFINED

**notDefined –** accessing a variable out of its scope

# Q Route level middleware

[**Our Route**](https://scotch.io/tutorials/route-middleware-to-check-if-a-user-is-authenticated-in-node-js#toc-our-route)

app.get('/hello', function(req, res) {

res.send('look at me!');

});

[Route Middleware Function](https://scotch.io/tutorials/route-middleware-to-check-if-a-user-is-authenticated-in-node-js#toc-route-middleware-function)

function isAuthenticated(req, res, next) {

// do any checks you want to in here// CHECK THE USER STORED IN SESSION FOR A CUSTOM VARIABLE

// you can do this however you want with whatever variables you set up

if (req.user.authenticated)

return next();

// IF A USER ISN'T LOGGED IN, THEN REDIRECT THEM SOMEWHERE

res.redirect('/');

}

## [**Implementing the Middleware**](https://scotch.io/tutorials/route-middleware-to-check-if-a-user-is-authenticated-in-node-js#toc-implementing-the-middleware)

Now that we have our function to check if our user is logged in or authenticated, we'll just apply it to our route.

app.get('/hello', isAuthenticated, function(req, res) {

res.send('look at me!');

});

# Q what are middlewares in nodeJs..importance

A Middleware is a callback that sits on top of the actual request handlers. It takes the same parameters as a route handler.

It has access to req and res object .. and can modify these objects it takes place before actual request handler

pre-processing of the request.

next() call inside a middleware invokes the next middleware or route handler depending on whichever is declared next. But next() call inside a route handler invokes the next route handler only. If there is a middleware next then it’s skipped. Therefore middlewares must be declared above all route handlers.

# Q What is iife in JavaScript and its importance

An IIFE is an anonymous function contained within a pair of parenthesis and is invoked immediately. The pair of parenthesis creates a local scope for all the code inside of it and makes the anonymous function a function expression

**IIFE**(Immediately Invoked Function Expression) is a [JavaScript](https://developer.mozilla.org/en-US/docs/Glossary/JavaScript) [function](https://developer.mozilla.org/en-US/docs/Glossary/function) that runs as soon as it is defined

It is a design pattern which is also known as [Self-Executing Anonymous Function](https://developer.mozilla.org/en-US/docs/Glossary/Self-Executing_Anonymous_Function) and contains two major parts. The first is the anonymous function with lexical scope enclosed within the [Grouping Operator](https://developer.mozilla.org/en-US/docs/Glossary/Grouping_Operator) (**function(){}();**). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.

The second part is creating the immediately executing function expression (), through which the JavaScript engine will directly interpret the function.

The primary reason to use an IIFE is to obtain data privacy. Because JavaScript's var scopes variables to their containing function, any variables declared within the IIFE cannot be accessed by the outside world.

IIFE produces a [lexical scope](https://en.wikipedia.org/wiki/Scope_(computer_science)) using JavaScript's [function scoping](https://en.wikipedia.org/wiki/Function_scoping). Immediately-invoked function expressions can be used to avoid [variable hoisting](https://en.wikipedia.org/wiki/JavaScript_syntax#Scoping_and_hoisting) from within blocks, protect against polluting the [global environment](https://en.wikipedia.org/wiki/Global_variable) and simultaneously allow public access to methods while retaining privacy for variables defined within the function.

**var** v, getValue;

v = 1;

getValue = **function** () { **return** v; };

v = 2;

getValue(); *// 2*

Hereafter the function passes v as an argument and is invoked immediately, preserving the inner function's execution context.[[14]](https://en.wikipedia.org/wiki/Immediately-invoked_function_expression#cite_note-JQ-14)

**var** v, getValue;

v = 1;

getValue = (**function** (x) {

**return** **function** () { **return** x; };

})(v);

v = 2;

getValue(); *// 1*

Purposes to use IIFE

***1 ) To Avoid Polluting the Global Scope***

Example without using IIFE : the value of counter in upCount function will 11 +1 = 2 as counter is taking previous value

var counter = 10;

counter++;

console.log('counter++ is '+counter); **//11**

var upCount = function(){

counter = counter + 1;

console.log('counter+1 is '+counter); **//12**

};

upCount();

Example withusing IIFE : the value of counter in upCount function will 11 +1 = 2 as counter is taking previous value . so to avoid populating global variable scope … wrap the scope in IIFE .

So IIFE will be executed as soon as it gets loaded.

**(function(){**

var counter = 10;

**})();**

var upCount = function(){

**var counter = 1;**

counter = counter + 1;

console.log('counter+1 is '+counter); **//2**

};

upCount();

***2 ) To Create Closures***

console.log('counter+1 is

***3) To limit the scope of variable***

If we try to access fo variable out of IIFE … it will give error …not defined . in this way we can limit the scope of variable

(function() {

var foo = "bar";

console.log(foo);

})();

foo; // ReferenceError: foo is not defined

***4) Establishing private variables and accessors***

IIFEs are also useful for establishing private methods for accessible functions while still exposing some properties for later use.[[17]](https://en.wikipedia.org/wiki/Immediately-invoked_function_expression#cite_note-17) The following example comes from Alman's post on IIFEs.[[1]](https://en.wikipedia.org/wiki/Immediately-invoked_function_expression#cite_note-Alman-1)

*// "counter" is a function that returns an object with properties, which in this case are functions.*

**var** counter = (**function** () {

**var** i = 0;

**return** {

get: **function** () { // this is closure also ..inner function

**return** i;

},

set: **function** (val) {

i = val;

},

increment: **function** () {

**return** ++i;

}

};

})();

*// These calls access the function properties returned by "counter".*

counter.get(); *// 0*

counter.set(3);

counter.increment(); *// 4*

counter.increment(); *// 5*

If we attempt to access counter.i from the global environment, it will be undefined, as it is enclosed within the invoked function and is not a property of counter. Likewise, if we attempt to access i, it will result in an error, as we have not declared i in the global environment.

So i scope is protected

# Option method

The **HTTP OPTIONS method** is used to describe the communication options for the target resource. The client can specify a specific URL for the OPTIONS method, or an asterisk (\*) to refer to the entire server.

OPTIONS /resources/post-here/ HTTP/1.1

Host: bar.other

Minimally, the response should be a 200 OK and have an Allow header with a list of HTTP methods that may be used on this resource. As an authorized user on an API, if you were to request OPTIONS /users/me, you should receive something like…

200 OK

Allow: HEAD,GET,PUT,DELETE,OPTIONS

# JSHint

JSHint is a static code analysis tool foris checking if JavaScript source code complies with coding rules

# Gulp

Gulp is a javascript task runner that lets you automate tasks such as…

* Bundling and minifying libraries and stylesheets.
* Refreshing your browser when you save a file.
* Quickly running unit tests
* Running code analysis
* Less/Sass to CSS compilation
* Copying modified files to an output directory

npm install --save-dev gulp

 devDependencies will need to be resolved at development time, where as dependencies will need to be resolved at run time. Because gulp is a tool to aid us in development, it needs to be resolved at development time.

#### **Creating a gulpfile**

A gulpfile is a file that will act as a manifest to define our tasks. Tasks that we want to execute will be found within this file. Whenever we run the command gulp hello-world from the command line, we are telling gulp that we want to run the hello-world task within gulpfile.js.

After creating gulpfile.js within the root of your project, add a basic tasks.

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

gulp.task('hello-world', **function**(){

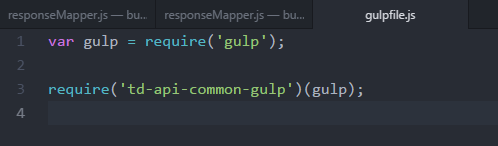
console.log('hello world');

});

require is a function implemented by node (which is an implementation of the [CommonJS spec](http://wiki.commonjs.org/wiki/Modules/1.1.1)) that will add references to node modules that we have installed. Once we make a reference to the gulp module, we can use it to create a task. Here, our task simply writes to the console window, but you could have it do any number of automated tasks.

 Task to analyze all of your JavaScript files inside of a set of directories with JSHint and JSCS may look something like this

Project == gulifile.js



td-api-common-gulp – index.js

module.exports = function (gulp) {

gulp.task('default', **defaultTask**);

gulp.task('test', **testTask**);

gulp.task('open-report', **openReportTask**); var gulpTasks = {

**testTask**: testTask,

**defaultTask**: defaultTask,

**openReportTask**: openReportTask,

};

return gulpTasks;

};

**Grunt** is a [JavaScript](https://en.wikipedia.org/wiki/JavaScript) [task runner](https://en.wikipedia.org/wiki/Build_tool), a tool used to automatically perform frequent tasks such as [minification](https://en.wikipedia.org/wiki/Minification_(programming)), [compilation](https://en.wikipedia.org/wiki/Compiler), [unit testing](https://en.wikipedia.org/wiki/Unit_testing), and [linting](https://en.wikipedia.org/wiki/Lint_(software)).

(Grunt like gulp)

# Mocha



# SUPERTEST:

|  |
| --- |
| const supertest = require('supertest') |
|  |  |
|  | describe('movies-service', () => { |
|  |  |
|  | const api = supertest('http://192.168.99.100:3000') |
|  |  |
|  | it('returns a 200 for a collection of movies', (done) => { |
|  |  |
|  | api.get('/movies/premiers') |
|  | .expect(200, done) |
|  | }) |
|  | }) |

# Prototype

***https://hackernoon.com/prototypes-in-javascript-5bba2990e04b***

var a;

a.name = “Shital”;

**Error** : cannot set property to undefined

var a = {};

a.name = “Shital”;

console.log(a.name); // output - Shital

will work

function Person(){

}

//Add property name, age to the prototype property of the Person constructor function

Person.prototype.name = "Ashwin" ;

Person.age = 26;

}

//Create an object using the Person constructor function

var person1 = new Person();

//Access the name property using the person object

console.log(person1.name)// Output" Ashwin

console.log(person1.age)// Output" undefined

JavaScript function has a **prototype property** (this property is empty by default), and you attach properties and methods on this prototype property when you want to implement inheritance.

All JavaScript objects inherit the properties and methods from their prototype.

Objects created using an object literal, or with new Object(), inherit from a prototype called Object.prototype.

The Object.prototype is on the top of the prototype chain.

1 \_\_Proto\_\_



This shows that person1’s dunder proto property and Human.prototype are pointing to the same object

Human.prototype === person1.\_\_proto\_\_ //true

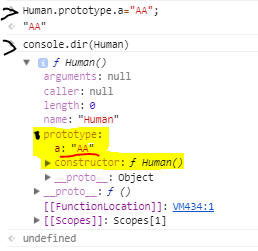
>> var person2=new Human();

Human.prototype === person2.\_\_proto\_\_ //true

person1.\_\_proto\_\_ === person2.\_\_proto\_\_ //true

**Prototype object of the constructor function is shared among all the objects created using the constructor function.**

2 property can be added to the constructor function’s prototype property



# Q Types of creating objects in JS

## 1 **Object literals**

var pasta = {

grain: "wheat",

width: 0.5,

shape: "round"

};

***2* Using a constructor function**

Alternatively, you can create an object with these two steps:

Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.

Create an instance of the object with new.

1. function Car(make, model, year) {
2. this.make = make;
3. this.model = model;
4. this.year = year;
5. }

Now you can create an object called mycar as follows:

var mycar = new Car('Eagle', 'Talon TSi', 1993);

### 3 Object.create method

Objects can also be created using the [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function.

// Animal properties and method encapsulation

var Animal = {

type: 'Invertebrates', // Default value of properties

displayType: function() { // Method which will display type of Animal

console.log(this.type);

}

};

// Create new animal type called animal1

var animal1 = Object.create(Animal);

animal1.displayType(); // Output:Invertebrates

# Shrink-Wrap

The issue arises due to the way npm install works. Since run­ning an npm install install a hier­ar­chy of pack­ages to be installed, if you wished to man­u­ally con­trol the ver­sion num­bers of the pack­ages that you want to be installed, you could do that by using the exact ver­sion num­bers in your package.json. How­ever that only solves the prob­lem for the direct depen­dents of your pack­age. It does not give you con­trol over the installed ver­sions of the deeply nested pack­ages that are the depen­den­cies of your depen­den­cies and beyond.

This can be cru­cial to you in a pro­duc­tion envi­ron­ment because you need to ensure that each pro­duc­tion re-deployment always always installs the same ver­sions of the pack­age as the other deployments.

This is where npm shrinkwrap comes into play. When you run npm shrinkwrap in a project after run­ning npm install, it cre­ates a file called npm-shrinkwrap.json which lists the exact pack­age ver­sions of all the installed pack­ages in the entire hier­ar­chy. If you check this into your ver­sion con­trol and your col­legue clones and does an npm install, then this time they will get the exact pack­age ver­sion for the full hier­ar­chy as spec­i­fied in the npm-shrinkwrap.json file.

# What is package-lock.json file in Node NPM?

When you are doing development in Angular, Node NPM is your tool for package management. In simple words, we have a “package.json” file and all dependencies are listed inside it. When you are doing NPM, you will always find “package-lock.json” file. So in this tutorial, we will unleash the importance of this lock file.

To understand the importance of lock, let's understand how software versioning works.

Most software versions follow semantic versioning. In semantic versioning, versions are divided into three distinct numbers as shown in the image below.

|  |  |
| --- | --- |
| The first number is termed as “major version”, second “minor version” and third “revision”.    **Major version**: Any increment in major version is an indication that there are breaking changes in the software functionality. It’s very much possible that the old code will not work with these changes and have to be tested properly.    **Minor version**: This version is incremented when we add new features, but the old code still works.    **Revision**: This version is incremented when we are just doing bug fixes. So there are no new functionalities added, no breaking changes and backward compatible with old code. | https://www.codeproject.com/KB/Nodejs/1202361/1.jpg |

|  |  |
| --- | --- |
| NPM follows semantic versioning, but it also has some more special characters like “^”, “~”, “>” and so on. They dictate how NPM get latest should behave for Major and Minor versions.    For these formats, 3 formats are very primary. Let’s understand each of them.    Exact (1.6.5), Major/Minor ( ^1.6.5) or Minor(~1.6.5). | https://www.codeproject.com/KB/Nodejs/1202361/2.jpg |

|  |  |
| --- | --- |
| **Exact (1.6.5)**: This will do a get latest of exact version 1.6.5 not more or not less. If that version is not available, it will throw up an exception.    **Major/Minor(^1.6.5)**: The carrot sign will get minimum 1.6.5 and if there are any higher MINOR / REVISION versions, it will get that. It WILL NEVER GET HIGHER MAJOR VERSIONS. So if 1.6.5 has 1.6.7 it will get that, if it has 1.7.7 it will that, but if it has 2.0 it will NOT get that.    **Minimum or lower (~1.6.5)**: The tilde sign will get HIGHER REVISIONS. For if 1.6.5 has 1.6.7 it will get that, but if it has 1.7.5 it will not be installed, if it has 2.0 it will not be installed. | https://www.codeproject.com/KB/Nodejs/1202361/3.jpg |

As discussed in the previous sections, package.json has “^” and “~” versioning mechanism. Now suppose in your package.json, you have mentioned "jquery": "^3.1.0"and Jquery has a new version “3.2.1”. So in actual, it will install or in other words, LOCK DOWN to “3.2.1”.

So in package.json, you will have “^3.1.0”, but actually you will be using “3.2.1”. This entry of actual version is present in “package-lock.json”. So package lock files have the EXACT versions which are used in your code.

[***https://medium.com/@Quigley\_Ja/everything-you-wanted-to-know-about-package-lock-json-b81911aa8ab8***](https://medium.com/@Quigley_Ja/everything-you-wanted-to-know-about-package-lock-json-b81911aa8ab8)

[***https://www.codeproject.com/Articles/1202361/What-is-package-lock-json-file-in-Node-NPM***](https://www.codeproject.com/Articles/1202361/What-is-package-lock-json-file-in-Node-NPM)

* NPM will automatically create a package-lock.json when installing packages unless there’s already npm-shrinkwrap.json in which case it will update it instead (if necessary).
* The new package-lock.json is never published and should be added to your version control system.
* Running npm shrinkwrap with a package-lock.json already present will just rename it to npm-shrinkwrap.json.
* When both files are present for some reason, package-lock.json will be ignored.

# What is the difference between tilde(~) and caret(^)

In the simplest terms, the tilde matches the most recent minor version (the middle number). ~1.2.3 will match all 1.2.x versions but will miss 1.3.0.

The caret, on the other hand, is more relaxed. It will update you to the most recent major version (the first number). ^1.2.3 will match any 1.x.x release including 1.3.0, but will hold off on 2.0.0

Npm allows installing newer version of a package than the one specified. Using tilde (~) gives you bug fix releases and caret (^) gives you backwards compatible new functionality as well.

The problem is old versions usually don't receive bug fixes that much, so npm uses caret (^) as the default for --save.

# Q production process manager

PM2 is a production process manager for Node.js / io.js applications with a built-in load balancer. It allows you to keep applications alive forever, to reload them without downtime and to facilitate common system admin tasks.

Starting an application in production mode is as easy as:

$ pm2 start app.js

## Install PM2

$ npm install pm2 -g

## Start an application

$ pm2 start app.js

Your app is now put in background, kept alive forever and monitored.

**Or** you can use pm2 **programmatically**:

var pm2 = require('pm2');

pm2.connect(function() {

pm2.start({

script : 'app.js', // Script to be run

exec\_mode : 'cluster', // Allow your app to be clustered

instances : 4, // Optional: Scale your app by 4

max\_memory\_restart : '100M' // Optional: Restart your app if it reaches 100Mo

}, function(err, apps) {

pm2.disconnect();

});

});

# When to use node

For network applications

1. If your server side code requires very few cpu cycles. In other world you are doing non blocking operation and does not have heavy algorithm/Job which consumes lots of CPU cycles.. that means node can not be used for cpu intensive applications

**Node**.**js** is a server-side JavaScript environment. It uses an asynchronous event-driven model and is designed for writing scalable internet applications,

Node.js, uses an event loop instead of threads. It is able to scale to millions of concurrent connections. Node.js takes advantage of the fact that servers spend most of their time waiting for I/O operations. Reading a file from a hard drive, accessing an external web service or waiting for a file to finish being uploaded, because these operations are much slower than in memory operations. Every I/O operation in Node.js is asynchronous. The server continues to process incoming requests while the I/O operation is taking place. JavaScript is well suited to event-based programming because it has anonymous functions and closures which make defining inline callbacks a cinch, and JavaScript developers already know how to program in this way. This event-based model makes Node.js very fast, and makes scaling real-time applications very easy.

Eval function ;

This is used to evaluate javaScript code.

If the argument is an expression, **eval**() evaluates the expression. If the argument is one or more JavaScript statements, **eval**() executes the statements

eval(new String('2 + 2')); // returns a String object containing "2 + 2"

eval('2 + 2'); // returns 4

# Q Difference between session and cookie

**session** data is stored on the server, whereas **cookies** store data in the visitor's browser.**Sessions** are more secure than **cookies** as it is stored in server.**Cookie** can be turn off from browser.

A cookie is a bit of data stored by the browser and sent to the server with every request.

A session is a collection of data stored on the server and associated with a given user (usually via a cookie containing an id code)

A session is a **set** of data that is stored on the server, usually as key-value pairs. A session is assigned a **pseudo**-**random**, secret ID that is usually stored in the user's browser using a cookie

# Q What is the js function keyword used to chk the datatype of variable

# How to differentiate the exact datatype of object and array

var a = [];

console.log('sasas '+ (a instanceof Array));

console.log('sasas '+ (a.constructor == Array));

console.log('sasas '+ Array.isArray(a));

all will return true

# **Q** What is strict mode

Sasas Strict mode is used for better error checking of the code …this is achieved by writing use strict at the top of the code.

# **Q** What are global in node js

These objects are available in all modules.

* [console](https://nodejs.org/api/globals.html#globals_console)
* [exports](https://nodejs.org/api/globals.html#globals_exports)
* [global](https://nodejs.org/api/globals.html#globals_global)
* [module](https://nodejs.org/api/globals.html#globals_module)
* [process](https://nodejs.org/api/globals.html#globals_process)
* [require()](https://nodejs.org/api/globals.html#globals_require)

# Q Explain process object

The process object is a global object and can be accessed from anywhere.

The process object is extremely useful for identifying information about the runtime environment of your node app such as the version of node, the arguments passed to the node executable, the current working directory, and the nextTick function.

The process object is an instance of EventEmitter and emits the following events

**1 Events - Exit, uncaughtException**

…Exit event…

Emitted when the process is about to exit. There is no way to prevent the exiting of the event loop at this point, and once all exit listeners have finished running, the process will exit.

Node.js process will exit immediately after calling the 'exit' event listeners and if any additional work still queued in the event loop will not be executed. In the following example, for instance, the timeout will never occur, event loop will never execute timeout function.

process.on('exit', function(code) {

// Following code will never execute.

setTimeout(function() {

console.log("This will not run");

}, 0);

console.log('About to exit with code:', code);

});

console.log("Program Ended");

Verify the Output.

Program Ended

About to exit with code: 0

Exit Codes

Node normally exits with a 0 status code when no more async operations are pending.

…**'uncaughtException'** event…

The 'uncaughtException' event is emitted when an uncaught JavaScript exception bubbles all the way back to the event loop. By default, Node.js handles such exceptions by printing the stack trace to stderr and exiting. Adding a handler for the 'uncaughtException' event overrides this default behavior.

process.on('uncaughtException', function(err) {

console.log(`This is exception `+err);

});

**2 Process provides many useful properties**

exiting. Addingstdout,stdin,stderr,execPath,pid,platform

// Printing to console

process.stdout.write("Hello World!" + "\n");

// Getting executable path

console.log(process.execPath);

// Platform Information

console.log(process.platform); //win32

**3 Process provides many useful methods**

cwd(), memoryUsage()

/ Print the current directory

console.log('Current directory: ' + process.cwd());

// Print the process version

console.log('Current version: ' + process.version);

// Print the memory usage

console.log(process.memoryUsage());

# Q Console statement .. print on both browser and node

**….which will print fast**.==(ANS node I think)

In a web browser(Chrome), declaring the variable i outside of any function scope makes it global and therefore binds to **window** object. As a result, running this code in a web browser requires repeatedly resolving the property within the heavily populated window namespace in each iteration of the for loop.

In **Node.js** however, declaring any variable outside of any function’s scope binds it only to the **module** scope (not the window object) which therefore makes it much easier and faster to resolve.

We will get more or less same execution speed when we wrap the above code in function

# Q-17. What Is EventEmitter In Node.Js?

**Answer.**

Events module in Node.js allows us to create and handle custom events. The Event module contains “EventEmitter” class which can be used to raise and handle custom events. It is accessible via the following code.

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

When an EventEmitter instance encounters an error, it emits an “error” event. When a new listener gets added, it fires a “newListener” event and when a listener gets removed, it fires a “removeListener” event.

EventEmitter provides multiple properties like “on” and “emit”. The “on” property is used to bind a function to the event and “emit” is used to fire an event.

# Can you access DOM in node?

No, you cannot access DOM in node.

WHY

Node.js is is a stand-alone JavaScript environment completely independent of a web browser. There's no link between web browsers and JavaScript; the DOM is **not** part of the JavaScript language or specification or anything.

There are projects like [jsdom](https://github.com/tmpvar/jsdom) if you want a virtual DOM in your Node project. Because of its very nature as a server-side platform, a DOM is a facility that Node can do without and still make perfect sense for a wide variety of server applications

# API Microgateway

The **API Microgateway** is a developer-focused, programmable **API** gateway written in **Node**.**js** to secure and control microservices and **APIs**. The gateway enables developers to quickly expose their existing services as **APIs** or create new **API** services from data sources, such as databases, without writing any code.

The microgateway is the enforcement component of the [**IBM API Connect**](https://developer.ibm.com/apiconnect/) collection of components providing solutions for API creation, deployment, lifecycle management, monitization, and enforcement. The microgateway is fundamentally a proxy, securing and forwarding requests to backend APIs.

It was created using StrongLoop technology and a series of middleware components. The package is customized to work with the apiconnect infrastructure that automatically communicates with the micro gateway to dynamically load APIs, Products, and Plans so that APIs are secured and processed in a seamless fashion.

The API Microgateway is a developer-focused, programmable API gateway written in Node.js to secure and control microservices and APIs. The gateway enables developers to quickly expose their existing services as APIs or create new API services from data sources, such as databases, without writing any code.

A gateway is used to protect and control access to the API services. These sets of capabilities are often related to security, such as enforcing API security (for example, OAuth) and controlling traffic (limiting the rate of transactions per seconds).

The API Microgateway provides a flexible flow engine that allows you to create flows to transform, enrich, or secure API traffic. Developers can create custom user policies that provide reusable policies to improve developer productivity and sharing across the developer community.

### What programming language or techniques will developers use or learn?

The API Microgateway is written using Node.js and is extensible using server-side JavaScript. The API definitions are stored in a Swagger 2.0 file, which allows developers to create new API definitions from existing Swagger 2.0 documents. Developers will learn how to use JavaScript to secure and control APIs and microservices that are defined using Swagger 2.0.

### Why should a developer contribute?

We want to provide a developer-friendly API microgateway platform that allows you to quickly meet your API and microservice requirements. It is built with an extensible plugin platform that enables you to add new capabilities, leveraging the depth of assets and expertise from the Node.js community.

### What technology problem will it help solve?

The API Microgateway simplifies API and microservices architectures, enabling developers to configure security and traffic management polices in a gateway that’s external to their API and microservices implementation. It enables API best practices by leveraging a gateway solution within their architecture to provide enhanced operational robustness and scalability.

### How will the API Microgateway help my business?

Businesses are driving digital innovation through the use of API services. The API Microgateway will accelerate the delivery of your API strategy, enabling easier sharing of business assets as APIs for rapidly delivering new digital services to increase brand awareness or potential monetization.

# STUB

Stubbing is the process of creating fake endpoints in code so that you can delay writing complex code, or to isolate what you are testing (see below re: mocks).

When testing a unit, often you’ll want to stub out that unit’s dependencies.

There are libraries that help us stub. Since we have a required module for eat-corn.js, there is one in particular that would do well for us called [proxyquire](https://github.com/thlorenz/proxyquire). It allows targeting the ./eat-corn.jsimport path and replacing it with your own module at test runtime. With a couple caveats, it usually works quite well. We’re going to not use it and see how we fare.

# Q How to communicate between one child to another

* Lets assume we have two Node.js processes running: example1.js and example2.js.
* In example1.js there is function func1(input) which returns result1 as a result.
* Is there a way from within example2.js to call func1(input) and obtain result1 as the outcome
* The simplest way to my knowledge, is to use [child\_process.fork()](http://nodejs.org/api/child_process.html#child_process_child_process_fork_modulepath_args_options):
* This is a special case of the spawn() functionality for spawning Node processes. In addition to having all the methods in a normal ChildProcess instance, the returned object has a communication channel built-in. The channel is written to with child.send(message, [sendHandle]) and messages are received by a 'message' event on the child.
* So, for your example, you could have example2.js:
* var fork = require('child\_process').fork;
* var example1 = fork(\_\_dirname + '/example1.js');
* example1.on('message', function(response) {
* console.log(response);
* });
* example1.send({func: 'input'});
* And example1.js:
* function func(input) {
* process.send('Hello ' + input);
* }
* process.on('message', function(m) {
* func(m);
* });

This will enable process.send() and process.on('message') in the child process. These can be used to communicate between the 2 processes.

|  |
| --- |
| if (process.send) { |
|  | process.send("Hello"); |
|  | } |
|  |  |
|  | process.on('message', message => { |
|  | console.log('message from parent:', message); |
|  | }); |

[**view raw**](https://gist.github.com/ndelangen/3b2b981a4795e51ef4f8cf583764eb8a/raw/0b06a0354d919148ee283bfe1162c550840c9fb6/child.js)[**child.js**](https://gist.github.com/ndelangen/3b2b981a4795e51ef4f8cf583764eb8a#file-child-js) hosted with ❤ by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | const fork = require('child\_process').fork; |
|  |  |
|  | const program = path.resolve('child.js'); |
|  | const parameters = []; |
|  | const options = { |
|  | stdio: [ 'pipe', 'pipe', 'pipe', 'ipc' ] |
|  | }; |
|  |  |
|  | const child = fork(program, parameters, options); |
|  | child.on('message', message => { |
|  | console.log('message from child:', message); |
|  | child.send('Hi'); |
|  | }); |

https://medium.com/@NorbertdeLangen/communicating-between-nodejs-processes-4e68be42b917

# Q Singleton = module.exports()

the **singleton pattern** is a [software design pattern](https://en.wikipedia.org/wiki/Software_design_pattern) that restricts the [instantiation](https://en.wikipedia.org/wiki/Instantiation_(computer_science)) of a [class](https://en.wikipedia.org/wiki/Class_(computer_programming)) to one [object](https://en.wikipedia.org/wiki/Object_(computer_science)). This is useful when exactly one object is needed to be shared across the system.

In most languages, sharing an object across your entire application can be a complex process.

n Node, however, this type of object is the default. Every module that you require is shared across your application, so there’s no need for any special classes or extra code.

### PRIVATE VS. PUBLIC

When creating a new module, the variables and functions you define are kept private by default. You’ll need to use [**module.exports**](http://nodejs.org/api/modules.html#modules_module_exports) to define your interface and make them public. This is a straight-forward process: whatever you set to module.exports becomes public, and is visible when your module is required.

// module.js

var a = 1; // Private

module.exports = {

b: 2 // Public

};

# Q how to increase NODE API security

**https://scotch.io/tutorials/authenticate-a-node-js-api-with-json-web-tokens**

### Using JWT token

To generate token

Config file

module.exports = {

'secret': 'ilovescotchyscotch',

'database': 'mongodb://noder:noderauth&54;proximus.modulusmongo.net:27017/so9pojyN'

};

* **secret**: used when we create and verify JSON Web Tokens

app.set('superSecret', config.secret);

const payload = {

admin: user.admin

};

var token = jwt.sign(payload, app.get('superSecret'), {

expiresInMinutes: 1440 // expires in 24 hours

});

Jwt.sign method creates jwt token using payload as admin = userId and secrete key

To verify token

jwt.verify(token, app.get('superSecret'), function(err, decoded) {

if (err) {

}

else{

Authenticated

}

Jwt.verify method :

The application can verify that the signature obtained from it’s own hashing operation matches the signature on the JWT itself (i.e. it matches the JWT signature created by the authentication server). If the signatures match, then that means the JWT is valid

JWT token looks like

JWT, we simply need to combine the components – (headers, payload, secrete), with periods (.) separating them. We use the base64url encoded versions of the header and of the payload, and the signature we arrived at in step 3.

Header

|  |
| --- |
| { |
|  | "typ": "JWT", |
|  | "alg": "HS256" |
|  | } |

“alg” key specifies which hashing algorithm is being used to create the JWT signature component

Payload

It, specifically the user ID.

The data inside the payload is referred to as the “claims” of the token.

|  |
| --- |
| { |
|  | "userId": "b08f86af-35da-48f2-8fab-cef3904660bd" |
|  | } |

Create the SIGNATURE

The signature is computed using the following pseudo code:

// signature algorithm

data = base64urlEncode( header ) + “.” + base64urlEncode( payload )

signature = Hash( data, **secret** );

// header  
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

// payload  
eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ

Then, using the joined encoded header and payload, and applying the specified signature algorithm(HS256) on the data string with the secret key set as the string “secret”, we get the following JWT Signature:

// signature

-xN\_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM

header.payload.signature structure

// JWT Token  
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN\_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM

 The data inside a JWT is **encoded** and **signed**, not **encrypted**. The purpose of encoding data is to transform the data’s structure. Signing data allows the data receiver to verify the authenticity of the source of the data. So encoding and signing data does NOT secure the data.

**Encoding** is for maintaining data usability and can be reversed by employing the same algorithm that **encoded** the content, i.e. no key is used. **Encryption** is for maintaining data confidentiality and requires the use of a key (kept secret) in order to return to plaintext.

### Using passport.js its npm package

[Passport](http://passportjs.org/) npm package.

Those articles used the session based authentication however, which has problems when we talk about scaling web services and creating an API that can be consumed across many devices and services

### Use Helmet

[Helmet](https://www.npmjs.com/package/helmet) can help protect your app from some well-known web vulnerabilities by setting HTTP headers appropriately.

Helmet is actually just a collection of nine smaller middleware functions that set security-related HTTP headers:

* [csp](https://github.com/helmetjs/csp) sets the Content-Security-Policy header to help prevent cross-site scripting attacks and other cross-site injections.
* [hidePoweredBy](https://github.com/helmetjs/hide-powered-by) removes the X-Powered-By header.
* [hpkp](https://github.com/helmetjs/hpkp) Adds [Public Key Pinning](https://developer.mozilla.org/en-US/docs/Web/Security/Public_Key_Pinning) headers to prevent man-in-the-middle attacks with forged certificates.
* [hsts](https://github.com/helmetjs/hsts) sets Strict-Transport-Security header that enforces secure (HTTP over SSL/TLS) connections to the server.
* [ieNoOpen](https://github.com/helmetjs/ienoopen) sets X-Download-Options for IE8+.
* [noCache](https://github.com/helmetjs/nocache) sets Cache-Control and Pragma headers to disable client-side caching.
* [noSniff](https://github.com/helmetjs/dont-sniff-mimetype) sets X-Content-Type-Options to prevent browsers from MIME-sniffing a response away from the declared content-type.
* [frameguard](https://github.com/helmetjs/frameguard) sets the X-Frame-Options header to provide [clickjacking](https://www.owasp.org/index.php/Clickjacking) protection.
* [xssFilter](https://github.com/helmetjs/x-xss-protection) sets X-XSS-Protection to enable the Cross-site scripting (XSS) filter in most recent web browsers.

Install Helmet like any other module:

$ npm install --save helmet

Then to use it in your code:

// ...

var helmet = require('helmet')

app.use(helmet())

// ...

### Tighten Session Cookies

Express has default cookie settings that aren’t highly secure. They can be manually tightened to enhance security - for both an application and its user.

* secret - A secret string for the cookie to be salted with.
* key: The name of the cookie - if left default (connect.sid), it can be detected and give away that an application is using Express as a web server.
* httpOnly - Flags cookies to be accessible by the issuing web server, which assists in preventing session hijacking.
* secure - Ensure that it is set to true - which requires TLS/SSL - to allow the cookie to **only** be used with HTTPS requests, and not insecure HTTP requests.
* domain - Indicates the specific domain that the cookie can be accessed from.
* path - indicates the path that the cookie is accepted on within an application's domain.
* expires - The expiration date of the cookie being set. Defaults to a session cookie. When setting a cookie, the application is storing data on the server. If a timely expiration is not set up on the cookie, the Express application could start consuming resources that would otherwise be free.

A basic example setup of how to use [express-session](https://github.com/expressjs/session) to securely set cookies:

const express = require('express');

const session = require('express-session');

const app = express();

app.use(session({

secret: 'mySecretCookieSalt',

key: 'myCookieSessionId',

cookie: {

httpOnly: true,

secure: true,

domain: 'example.com',

path: '/foo/bar',

// Cookie will expire in 1 hour from when it's generated

expires: new Date( Date.now() + 60 \* 60 \* 1000 )

}

}));

# Q this in javaScript

 it refers to an object; that is, the subject in context, or the subject of the executing code. The this keyword not only refers to the object but it also contains the value of the object.

example:

|  |  |
| --- | --- |
|  | var person = { |
|  | firstName: "Penelope", |
|  | lastName: "Barrymore", |
|  | fullName: function () { |
|  | ​// Notice we use "this" just as we used "he" in the example sentence earlier?:​ |
|  | console.log(this.firstName + " " + this.lastName); |
|  | ​// We could have also written this:​​ |
|  | console.log(person.firstName + " " + person.lastName); |
|  | } |
|  | }  If we use person.firstName and person.lastName, as in the last example, our **code becomes ambiguous**. Consider that there could be another global variable (that we might or might not be aware of) with the name “person.” Then, references to person.firstName **could attempt to access the firstName property from the person global variable**, and this could lead to difficult-to-debug errors. So we use the “this” keyword not only for aesthetics (i.e., as a referent), but also for precision; its use actually makes our code more unambiguous, just as the pronoun “he” made our sentence more clear. It tells us that we are referring to the specific John at the beginning of the sentence. |

all functions in JavaScript have properties, just as objects have properties. And when a function executes, it gets the this property

The *this* reference ALWAYS refers to (and holds the value of) an object—a singular object—and it is usually used inside a function or a method, although it can be used outside a function in the global scope. Note that when we use strict mode, *this* holds the value of *undefined* in global functions and in anonymous functions that are not bound to any object.  
  
*this* is used inside a function (let’s say function A) and it contains the value of the object that invokes function A. We need *this* to access methods and properties of the object that invokes function A, especially since we don’t always know the name of the invoking object, and sometimes there is no name to use to refer to the invoking object. Indeed, *this* is really just a shortcut reference for the “antecedent object”—the invoking object.

Ruminate on this basic example illustrating the use of *this* in JavaScript:

|  |  |
| --- | --- |
|  | var person = { |
|  | firstName :"Penelope", |
|  | lastName :"Barrymore", |
|  | // Since the "this" keyword is used inside the showFullName method below, and the showFullName method is defined on the person object,​ |
|  | // "this" will have the value of the person object because the person object will invoke showFullName ()​ |
|  | showFullName:function () { |
|  | console.log (this.firstName + " " + this.lastName); |
|  | } |
|  | ​ |
|  | } |
|  | ​ |
|  | person.showFullName (); // Penelope Barrymore |

## The use of **this** in the global scope

In the global scope, when the code is executing in the browser, all global variables and functions are defined on the window object. Therefore, when we use this in a global function, it refers to (and has the value of) the global window object (not in strict mode though, as noted earlier) that is the main container of the entire JavaScript application or web page.

Thus:

|  |  |
| --- | --- |
|  | var firstName = "Peter", |
|  | lastName = "Ally"; |
|  | ​ |
|  | function showFullName () { |
|  | // "this" inside this function will have the value of the window object​ |
|  | // because the showFullName () function is defined in the global scope, just like the firstName and lastName​ |
|  | console.log (this.firstName + " " + this.lastName); |
|  | } |
|  | ​ |
|  | var person = { |
|  | firstName :"Penelope", |
|  | lastName :"Barrymore", |
|  | showFullName:function () { |
|  | // "this" on the line below refers to the person object, because the showFullName function will be invoked by person object.​ |
|  | console.log (this.firstName + " " + this.lastName); |
|  | } |
|  | } |
|  | ​ |
|  | showFullName (); // Peter Ally​ |
|  | ​ |
|  | // window is the object that all global variables and functions are defined on, hence:​ |
|  | window.showFullName (); // Peter Ally​ |
|  | ​ |
|  | // "this" inside the showFullName () method that is defined inside the person object still refers to the person object, hence:​ |
|  | person.showFullName (); // Penelope Barrymore |

**Context of this :**

1. global

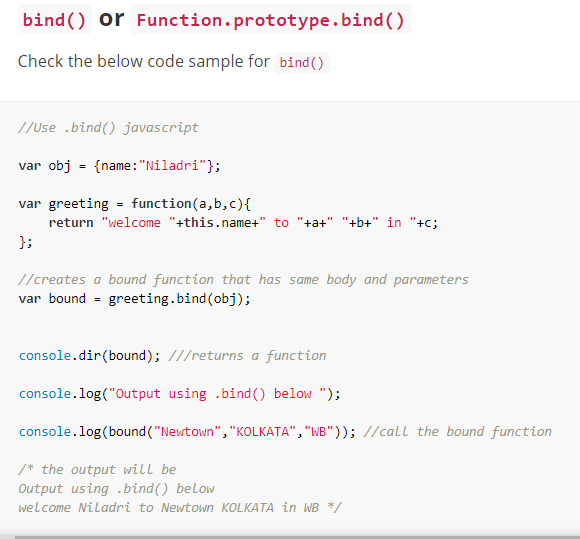
outside of any function , this refers to the global object whether in strict mode or not.

1. Functional

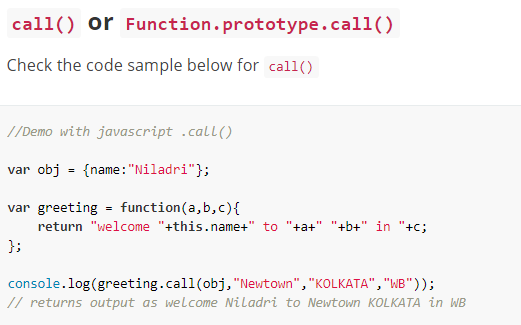
Inside a function, the value of this depends on how the function is called.

# Bind() call() apply()

***Bind method is used to bind a this value to variable***



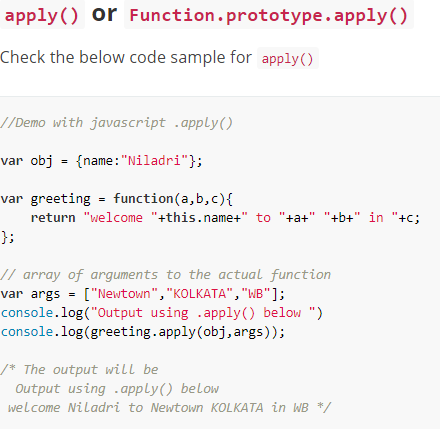
In the above code sample for bind() we are returning a bound function with the context which will be invoked later. We can see the bound function in the console as below .



You can use call()/apply() to invoke the function immediately.

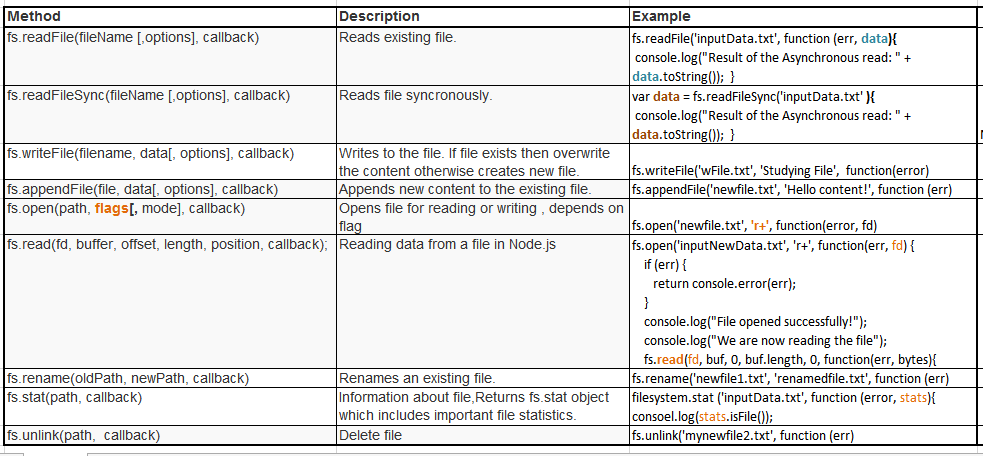
The first parameter in call() method sets the **"this"** value, which is the object, on which the function is invoked upon. In this case, it's the **"obj"** object above.

The rest of the parameters are the arguments to the actual function.



Similarly to call() method the first parameter in apply() method sets the **"this"**value which is the object upon which the function is invoked. In this case it's the **"obj"** object above. The only difference of apply() with the call() method is that the second parameter of the apply() method accepts the arguments to the actual function as an array.

# File system



|  |
| --- |
| fs.ftruncate(fd, len, callback)   * **fd** - This is the file descriptor returned by file fs.open() method. * **len** - This is the length of the file after which file will be truncated.  Difference between readFile and readFileSync [fs.readFile](http://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback) takes a call back ,  [fs.readFileSync](http://nodejs.org/api/fs.html#fs_fs_readfilesync_filename_options), does not take a callback if you give cb function in Sync your callback function will may be give exception or never get called and therefore the response will never end and it will timeout.  you should never call readFileSync in a node express/webserver since it will tie up the single thread loop while I/O is performed. You want the node loop to process other requests until the I/O completes and your callback handling code can run.  It's fine to call readFileSync to load your SSL certificates from disk as the server starts, as you need to block until those are ready |

var fs = require('fs');

console.log('...Start ...');

fs.writeFile('myFileWrite3.txt','file333333',function(err,data){

if(err){

console.log('errorrrr'+err);

}else{

console.log('write successfullll..'+data);

}

});

fs.readFile('myFileWrite.txt',function(err,data){

if(err){

console.log('errorrrr'+err);

}else{

console.log(' Read async..'+data);

}

});

var data = fs.readFileSync('myFileWrite2.txt');

console.log('Read Sync..'+data);

console.log('...End ...');

output

**Async will print at last…**

...Start ...

Read Sync..file2222222222222

...End ...

Read async..Hello

write successfullll..undefinedStreams

file system is important to read files like index.html and get there data in to code

we can also read certificates using fs

# What is the purpose of \_\_filename variable?

The \_\_filename represents the filename of the code being executed. This is the resolved absolute path of this code file. For a main program this is not necessarily the same filename used in the command line. The value inside a module is the path to that module file.

### What is the purpose of \_\_dirname variable?

The \_\_dirname represents the name of the directory that the currently executing script resides in.

# **Streams**

Streams are used to handel more data (100GB) . we can get this data in chunks with the help of streams . Instead of loading the whole file into memory, the system will the log file in chunks

<https://www.sitepoint.com/basics-node-js-streams/>

* **Readable** − Stream which is used for read operation.
* **Writable** − Stream which is used for write operation.
* **Duplex** − Stream which can be used for both read and write operation.
* **Transform** − A type of duplex stream where the output is computed based on input.

Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times.

we can connect chunks of code, manipulate the data in a very specific way, and pass it to the next piece of code

Using streams created by Node.js core modules, the data type for chunk will usually be a [Buffer](http://nodejs.org/api/buffer.html) or a string, both of which implement the length method.

## **Readable Stream**

A readable stream lets you read data from a source. The source can be anything. It can be a simple file on your file system, a buffer in memory or even another stream

### Reading From Streams

var fs = require('fs');

var readableStream = fs.createReadStream('file.txt');

var data = '';

readableStream.on('data', function(chunk) {

data+=chunk;

});

readableStream.on('end', function() {

console.log(data);

});

The function call fs.createReadStream() gives you a readable stream. Initially, the stream is in a static state. As soon as you listen to data event and attach a callback it starts flowing. After that, chunks of data are read and passed to your callback.

When there is no more data to read (end is reached), the stream emits an end event.

readable event is emitted when a chunk of data can be read from the stream .

non-flowing mode, the stream pushes some of it’s data to the read queue and then emits its readable event.

When we receive the readable event, we know our stream has data in it’s buffer that’s available to be read, and start consuming the data by calling read() on the stream.

We can get a better sense of what’s going on from following the output. The stream pushes two objects to the read queue and then fires the readableevent. Once the consumer starts reading objects, it frees up room in the read queue for our stream to continue pushing objects to. When the stream is done, and the read queue is empty, the end event is emitted.

Flow mode - .on(‘data’) event

Non-Flow mode - .on(‘redable) event

*https://www.sandersdenardi.com/readable-writable-transform-streams-node/*

There is also another way to read from stream. You just need to call read() on the stream instance repeatedly until every chunk of data has been read.

var fs = require('fs');

var readableStream = fs.createReadStream('file.txt');

var data = '';

var chunk;

readableStream.on('readable', function() {

while ((chunk=readableStream.read()) != null) {

data += chunk;

}

});

readableStream.on('end', function() {

console.log(data)

});

The read() function reads some data from the internal buffer and returns it. When there is nothing to read, it returns null. So, in the while loop we check for null and terminate the loop.

### Setting Encoding

readableStream.setEncoding('utf8');

In the above snippet we set the encoding to utf8. As a result, the data is interpreted as utf8 and passed to your callback as string.

### Additional methods

1. Readable.pause() – This method pauses the stream. If the stream is already flowing, it won’t emit data events anymore. The data will be kept in buffer. If you call this on a static (non-flowing) stream, the stream starts flowing, but data events won’t be emitted.
2. Readable.resume() – Resumes a paused stream.
3. readable.unpipe() – This removes destination streams from pipe destinations. If an argument is passed, it stops the readable stream from piping into the particular destination stream. Otherwise, all the destination streams are removed.

## **Writable Streams**

Writable streams let you write data to a destination

var fs = require('fs');

var readableStream = fs.createReadStream('file1.txt');

var writableStream = fs.createWriteStream('file2.txt');

readableStream.setEncoding('utf8');

readableStream.on('data', function(chunk) {

writableStream.write(chunk);

});

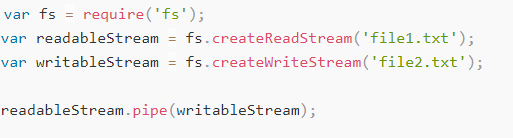
This function returns a Boolean value indicating if the operation was successful. If true, then the write was successful and you can keep writing more data. If false is returned, it means something went wrong and you can’t write anything at the moment. The writable stream will let you know when you can start writing more data by emitting a drainevent.

every chunk of data has been flushed, a finish event is emitted by the stream.

1. error – Emitted to indicate that an error has occurred while writing/piping.
2. pipe – When a readable stream is piped into a writable stream, this event is emitted by the **writable stream**.
3. unpipe – Emitted when you call unpipe on the readable stream and stop it from piping into the destination stream.



## **Pipe**



inp.pipe(gzip).pipe(out);

It does like…input.pipe(compressioTool).outPut

This compresses the input and sets it to output in one line

Gzip is compression algo

## **Chaining**



### Use Case of a Stream in Node.js

Imagine that you have two hard drives, one with a 100GB file and another with enough space to hold the output. Let's assume that the file is a log file, where we want to extract some useful data.

Loading 100GB into memory on your laptop would not be feasible, but we can solve the problem with Streams. Instead of loading the whole file into memory, the system will load the log file in chunks. This allows your app to use a constant amount of RAM.

Basically, your laptop is just a proxy that manipulates the data and dumps the result in another place, thus making it possible to do work that otherwise would be impossible.

[*https://www.codementor.io/davidgatti/understanding-streams-in-node-js-weupiug0a#im-reading-a-file-as-a-stream-so-i-must-know-the-size-right*](https://www.codementor.io/davidgatti/understanding-streams-in-node-js-weupiug0a#im-reading-a-file-as-a-stream-so-i-must-know-the-size-right)

# Mongoose

Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment.

Mongoose is an object modeling package for Node that essentially works like an ORM- object-relational mapping

• Every model has an associated connection (this will be the default connection when you use mongoose.model()). You create a new connection and call .model() on it to create the documents on a different database.

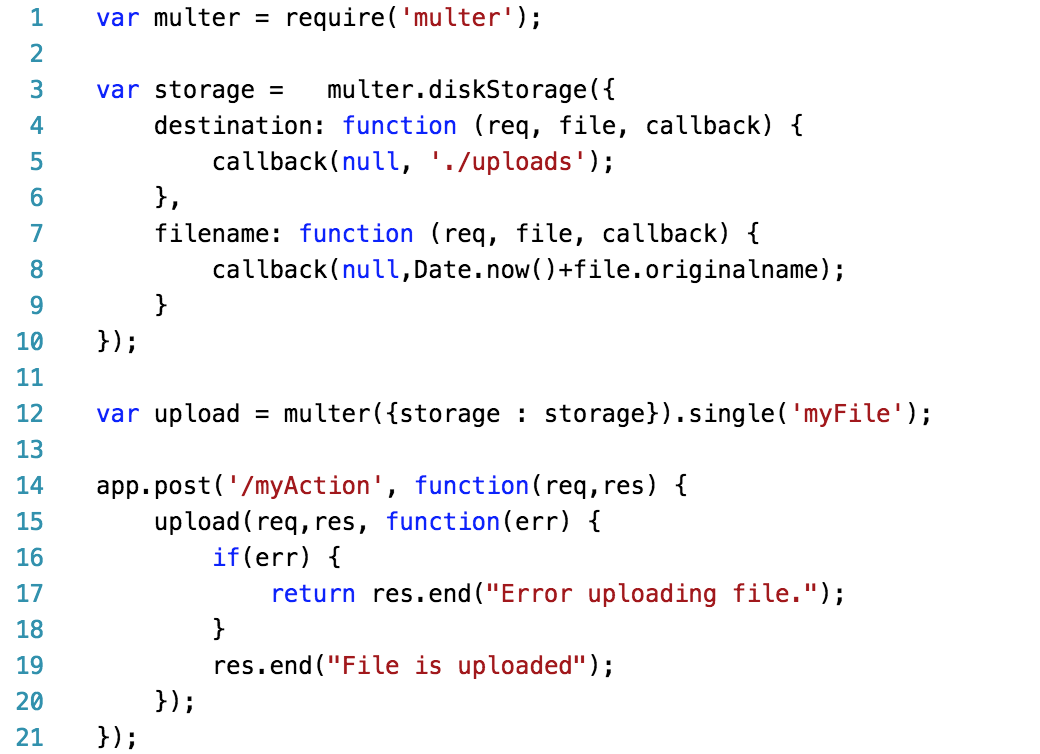
• You can access the fields in this new record using the dot syntax, and change the values. You have to call save() or update() to store modified values back to the database.

# **Istanbul:**

A Javascript code coverage tool written in JS

**statement, branch, and function, line coverage**

# Multer



var upload **=** multer({ dest**:** 'uploads/' })

If you want more control over your uploads, you'll want to use the storage option instead of dest. Multer ships with storage engines DiskStorage and MemoryStorage; More engines are available from third parties.

var storage **=** multer.diskStorage({

  destination**:** function (req, file, cb) {

    cb(null, '/tmp/my-uploads')

  },

  filename**:** function (req, file, cb) {

    cb(null, file.fieldname **+** '-' **+** Date.now())

  }

})

var upload **=** multer({ storage**:** storage })

destination is used to determine within which folder the uploaded files should be stored. This can also be given as a string (e.g. '/tmp/uploads'). If no destination is given, the operating system's default directory for temporary files is used.

**Note:** You are responsible for creating the directory when providing destination as a function. When passing a string, multer will make sure that the directory is created for you.

filename is used to determine what the file should be named inside the folder. If no filename is given, each file will be given a random name that doesn't include any file extension.

**Note:** Multer will not append any file extension for you, your function should return a filename complete with an file extension.

#### .single(fieldname)

Accept a single file with the name fieldname. The single file will be stored in req.file.

#### .array(fieldname[, maxCount])

Accept an array of files, all with the name fieldname. Optionally error out if more than maxCount files are uploaded. The array of files will be stored in req.files.