**Q. 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!");

})

**Q. Saving XML filen in Node.js [duplicate]**

*DOM implementations like XMLDOM have an XMLSerializer class (Most browsers have, too). This is part of the DOM API standard.*

*It allows to serialize a DOM document/node into an string. You can use the fs module to save the string in a file.*

var fs = require('fs');

var serializer = new (require('xmldom')).XMLSerializer;

var implementation = new (require('xmldom')).DOMImplementation;

var document = implementation.createDocument('', '', null);

document.appendChild(document.createElement('foo'));

fs.writeFile(

"/tmp/test.xml",

serializer.serializeToString(document),

function(error) {

if (error) {

console.log(error);

} else {

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

}

}

);

var myVeryLongInternalName = function() { ... };

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:

var m = require('./mymodule');

m.shortName();

**Q. module.exports is the object that's actually returned as the result of a require call.**

**The exports variable is initially set to that same object (i.e. it's a shorthand "alias"), so in the module code you would usually write something like this:**

var myFunc1 = function() { ... };

var myFunc2 = function() { ... };

exports.myFunc1 = myFunc1;

exports.myFunc2 = myFunc2;

to export (or "expose") the internally scoped functions myFunc1 and myFunc2.

And in the calling code you would use:

var m = require('./mymodule');

m.myFunc1();

where the last line shows how the result of require is (usually) just a plain object whose properties may be accessed.

NB: if you overwrite exports then it will no longer refer to module.exports. So if you wish to assign a new object (or a function reference) to exports then you should also assign that new object to module.exports

It's worth noting that the name added to the exports object does not have to be the same as the module's internally scoped name for the value that you're adding, so you could have:

var myVeryLongInternalName = function() { ... };

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:

var m = require('./mymodule');

m.shortName();

**Q What does “use strict” do in JavaScript, and what is the reasoning behind it?**

*This article about Javascript Strict Mode might interest you: John Resig - ECMAScript 5 Strict Mode, JSON, and Morevar myVeryLongInternalName = function() { ... };*

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:

var m = require('./mymodule');

*m.shortName();*

*To quote some interesting parts:*

*Strict Mode is a new feature in ECMAScript 5 that allows you to place a program, or a function, in a "strict" operating context. This strict context prevents certain actions from being taken and throws more exceptions.*

*It catches some common coding bloopers, throwing exceptions.*

*It prevents, or throws errors, when relatively "unsafe" actions are taken (such as gaining access to the global object).*

*It disables features that are confusing or poorly thought out.*

*And:*

var myVeryLongInternalName = function() { ... };

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:

var m = require('./mymodule');

*m.shortName();*

*Strict mode helps out in a couple ways:*

var myVeryLongInternalName = function(var myVeryLongInternalName = function() { ... };var myVeryLongInternalName = function() { ... };

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:

var m = require('./mymodule');

m.shortName();

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:var myVeryLongInternalName = function() { ... };

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:

var m = require('./mymodule');

m.shortName();

var m = require('./mymodule');

m.shortName();) { ... };

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:

var m = require('./mymodule');

*m.shortName();*

*It catches some common coding bloopers, throwing exceptions.var myVeryLongInternalName = function() { ... };*

exports.shortName = myVeryLongInternalName;

// add other objects, functions, as required

followed by:

var m = require('./mymodule');

*m.shortName();*

*It prevents, or throws errors, when relatively "unsafe" actions are taken (such as gaining access to the global object).*

*It disables features that are confusing or poorly thought out.*

*Also note you can apply "strict mode" to the whole file... Or you can use it only for a specific function (still quoting from John Resig's article):*

// Non-strict code...

(function(){

"use strict";

// Define your library strictly...

})();

// Non-strict code...

*Which might be helpful if you have to mix old and new code ;-)*

*So, I suppose it's a bit like the "use strict" you can use in Perl (hence the name?): it helps you make fewer errors, by detecting more things that could lead to breakages.*

*Currently, it's supported by all major browsers (bar IE 9 and below).*

***Q. Cluster***

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);

console.log(`Worker ${process.pid} started`);

}

With node.js, module.exports is how one exposes the public interface of a module.

/\* my-module.js \*/

exports.coolFunction = function(callback) {

// stuff & things

callback(whatever);

};

This interface can then be consumed by another module after importing/requiring it:

/\* another-module.js \*/

var myModule = require('my-module');

myModule.coolFunction(function(error) { ... });

Prototypes (a plain Javascript feature), on the other hand, are useful for defining shared properties and methods of objects instantiated from a constructor function.

function User() {

this.name = null;

}

User.prototype.printGreeting = function() {

console.log('Hello. My name is: ' + this.name);

};

var user = new User();

user.name = 'Jill';

user.printGreeting();

Cheers.

***Q. Javascript Modules: Prototype vs. Export***

**Q. Common method for calling a node function from Angular Controller**

*First of all nodejs and angularjs all though both are javascript both are two different implementation.*

*NodeJS works on server, on other hand angularjs works on browser.*

*initially when i was newbie to node i was also having the same problem . i was thinking we could directly call the node function from angularjs ,after all everything is javascript right ! but i was wrong.*

**Q. Common method for calling a node function from Angular Controller**

function MyCtrl($scope, $http) {

// $http is injected by angular's IOC implementation

// other functions and controller stuff is here...

// this is called when button is clicked

$scope.batchfile = function() {

$http.get('/performbatch').success(function() {

// url was called successfully, do something

// maybe indicate in the UI that the batch file is

// executed...

});

}

}

**How do I remove a property from a JavaScript object?**

delete myObject.regex;

// or,

delete myObject['regex'];

// or,

var prop = "regex";

delete myObject[prop];

Demo

var myObject = {

"ircEvent": "PRIVMSG",

"method": "newURI",

"regex": "^http://.\*"

};

delete myObject.regex;

console.log(myObject);

**Q. Node.js console.log() in txt file**

would use a library instead of re-inventing the wheel. I looked for a log4j-type library on npm, and it came up with https://github.com/nomiddlename/log4js-node

if you want to log to the console and to a file:

var log4js = require('log4js');

log4js.configure({

appenders: [

{ type: 'console' },

{ type: 'file', filename: 'logs/cheese.log', category: 'cheese' }

]

});

now your code can create a new logger with

var logger = log4js.getLogger('cheese');

and use the logger in your code

logger.warn('Cheese is quite smelly.');

logger.info('Cheese is Gouda.');

logger.debug('Cheese is not a food.');

**Q.Upgrading Node.js to latest version**

The module n makes version-management easy:

sudo npm install n -g

For the latest stable version:

sudo n stable

For the latest version:

sudo n latest

**How do I update Node.js?**

*Use Node Version Manager (NVM)*

*It's a Bash script that lets you download and manage different versions of node. Full source code is here.*

*There is a separate project for nvm for Windows: github.com/coreybutler/nvm-windows*