[Mohamed-Hashem](https://github.com/Mohamed-Hashem) / [**nodejs-interview-questions**](https://github.com/Mohamed-Hashem/nodejs-interview-questions)

**Q. What is Node.js?**

Node.js is an open-source server side runtime environment built on Chrome's V8 JavaScript engine. It provides an event driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side applications using JavaScript.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is Node.js Process Model?**

Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms.

All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request. So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are the key features of Node.js?**

* **Asynchronous and Event driven** – All APIs of Node.js are asynchronous. This feature means that if a Node receives a request for some Input/Output operation, it will execute that operation in the background and continue with the processing of other requests. Thus it will not wait for the response from the previous requests.
* **Fast in Code execution** – Node.js uses the V8 JavaScript Runtime engine, the one which is used by Google Chrome. Node has a wrapper over the JavaScript engine which makes the runtime engine much faster and hence processing of requests within Node.js also become faster.
* **Single Threaded but Highly Scalable** – Node.js uses a single thread model for event looping. The response from these events may or may not reach the server immediately. However, this does not block other operations. Thus making Node.js highly scalable. Traditional servers create limited threads to handle requests while Node.js creates a single thread that provides service to much larger numbers of such requests.
* **Node.js library uses JavaScript** – This is another important aspect of Node.js from the developer's point of view. The majority of developers are already well-versed in JavaScript. Hence, development in Node.js becomes easier for a developer who knows JavaScript.
* **There is an Active and vibrant community for the Node.js framework** – The active community always keeps the framework updated with the latest trends in the web development.
* **No Buffering** – Node.js applications never buffer any data. They simply output the data in chunks.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain how does Node.js work?**

A Node.js application creates a single thread on its invocation. Whenever Node.js receives a request, it first completes its processing before moving on to the next request.

Node.js works asynchronously by using the event loop and callback functions, to handle multiple requests coming in parallel. An Event Loop is a functionality which handles and processes all your external events and just converts them to a callback function. It invokes all the event handlers at a proper time. Thus, lots of work is done on the back-end, while processing a single request, so that the new incoming request doesn't have to wait if the processing is not complete.

While processing a request, Node.js attaches a callback function to it and moves it to the back-end. Now, whenever its response is ready, an event is called which triggers the associated callback function to send this response.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 2. NODE.JS SETUP**

**Q. How to create a simple server in Node.js that returns Hello World?**

**Step 01**: Create a project directory

mkdir myapp

cd myapp

**Step 02**: Initialize project and link it to npm

npm init

This creates a package.json file in your myapp folder. The file contains references for all npm packages you have downloaded to your project. The command will prompt you to enter a number of things. You can enter your way through all of them EXCEPT this one:

entry point: (index.js)

Rename this to:

app.js

**Step 03**: Install Express in the myapp directory

npm install express --save

**Step 04**: app.js

/\*\*

\* Express.js

\*/

const express = require('express');

const app = express();

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

res.send('Hello World!');

});

app.listen(3000, function () {

console.log('App listening on port 3000!');

});

**Step 05**: Run the app

node app.js

**⚝** [**Try this example on CodeSandbox**](https://codesandbox.io/s/hello-world-in-nodejs-ue3cs3)

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain the concept of URL module in Node.js?**

The URL module in Node.js splits up a web address into readable parts. Use require() to include the module. Then parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties.

**Example:**

/\*\*

\* URL Module in Node.js

\*/

const url = require('url');

const adr = 'http://localhost:8080/default.htm?year=2022&month=september';

const q = url.parse(adr, true);

console.log(q.host); // localhost:8080

console.log(q.pathname); // "/default.htm"

console.log(q.search); // "?year=2022&month=september"

const qdata = q.query; // { year: 2022, month: 'september' }

console.log(qdata.month); // "september"

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 3. NODE.JS DATA TYPES**

**Q. What are the data types in Node.js?**

Just like JS, there are two categories of data types in Node: Primitives and Objects.

**1. Primitives:**

* String
* Number
* BigInt
* Boolean
* Undefined
* Null
* Symbol

**2. Objects:**

* Function
* Array
* Buffer

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain String data type in Node.js?**

Strings in Node.js are sequences of unicode characters. Strings can be wrapped in a single or double quotation marks. Javascript provide many functions to operate on string, like indexOf(), split(), substr(), length.

**String functions:**

| **Function** | **Description** |
| --- | --- |
| charAt() | It is useful to find a specific character present in a string. |
| Concat() | It is useful to concat more than one string. |
| indexOf() | It is useful to get the index of a specified character or a part of the string. |
| Match() | It is useful to match multiple strings. |
| Split() | It is useful to split the string and return an array of string. |
| Join() | It is useful to join the array of strings and those are separated by comma (,) operator. |

**Example:**

/\*\*

\* String Data Type

\*/

const str1 = "Hello";

const str2 = 'World';

console.log("Concat Using (+) :" , (str1 + ' ' + str2));

console.log("Concat Using Function :" , (str1.concat(str2)));

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain Number data type in Node.js?**

The number data type in Node.js is 64 bits floating point number both positive and negative. The parseInt() and parseFloat() functions are used to convert to number, if it fails to convert into a number then it returns NaN.

**Example:**

/\*\*

\* Number Data Type

\*/

// Example 01:

const num1 = 10;

const num2 = 20;

console.log(`sum: ${num1 + num2}`);

// Example 02:

console.log(parseInt("32")); // 32

console.log(parseFloat("8.24")); // 8.24

console.log(parseInt("234.12345")); // 234

console.log(parseFloat("10")); // 10

// Example 03:

console.log(isFinite(10/5)); // true

console.log(isFinite(10/0)); // false

// Example 04:

console.log(5 / 0); // Infinity

console.log(-5 / 0); // -Infinity

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain BigInt data type in Node.js?**

A BigInt value, also sometimes just called a BigInt, is a bigint primitive, created by appending **n** to the end of an integer literal, or by calling the BigInt() function ( without the new operator ) and giving it an integer value or string value.

**Example:**

/\*\*

\* BigInt Data Type

\*/

const maxSafeInteger = 99n; // This is a BigInt

const num2 = BigInt('99'); // This is equivalent

const num3 = BigInt(99); // Also works

typeof 1n === 'bigint' // true

typeof BigInt('1') === 'bigint' // true

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain Boolean data type in Node.js?**

Boolean data type is a data type that has one of two possible values, either true or false. In programming, it is used in logical representation or to control program structure.

The boolean() function is used to convert any data type to a boolean value. According to the rules, false, 0, NaN, null, undefined, empty string evaluate to false and other values evaluates to true.

**Example:**

/\*\*

\* Boolean Data Type

\*/

// Example 01:

const isValid = true;

console.log(isValid); // true

// Example 02:

console.log(true && true); // true

console.log(true && false); // false

console.log(true || false); // true

console.log(false || false); // false

console.log(!true); // false

console.log(!false); // true

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain Undefined and Null data type in Node.js?**

In node.js, if a variable is defined without assigning any value, then that will take **undefined** as value. If we assign a null value to the variable, then the value of the variable becomes **null**.

**Example:**

/\*\*

\* NULL and UNDEFINED Data Type

\*/

let x;

console.log(x); // undefined

let y = null;

console.log(y); // null

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain Symbol data type in Node.js?**

Symbol is an immutable primitive value that is unique. It's a very peculiar data type. Once you create a symbol, its value is kept private and for internal use.

**Example:**

/\*\*

\* Symbol Data Type

\*/

const NAME = Symbol()

const person = {

[NAME]: 'Ritika Bhavsar'

}

person[NAME] // 'Ritika Bhavsar'

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain function in Node.js?**

Functions are first class citizens in Node's JavaScript, similar to the browser's JavaScript. A function can have attributes and properties also. It can be treated like a class in JavaScript.

**Example:**

/\*\*

\* Function in Node.js

\*/

function Messsage(name) {

console.log("Hello "+name);

}

Messsage("World"); // Hello World

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain Buffer data type in Node.js?**

Node.js includes an additional data type called Buffer ( not available in browser's JavaScript ). Buffer is mainly used to store binary data, while reading from a file or receiving packets over the network.

**Example:**

/\*\*

\* Buffer Data Type

\*/

let b = new Buffer(10000);

let str = "----------";

b.write(str);

console.log( str.length ); // 10

console.log( b.length ); // 10000

*Note: Buffer() is deprecated due to security and usability issues.*

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

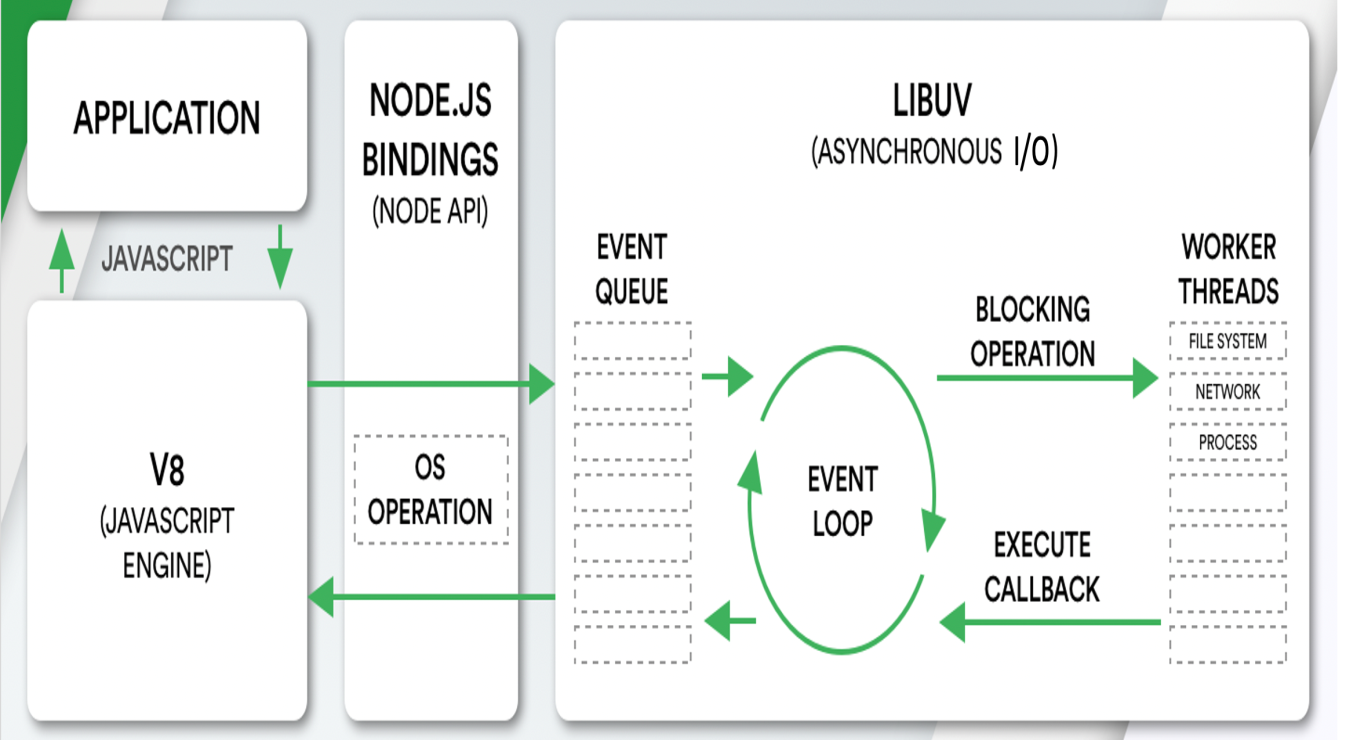
**# 4. NODE.JS ARCHITECTURE**

**Q. How does Node.js works?**

Node.js is completely event-driven. Basically the server consists of one thread processing one event after another.

A new request coming in is one kind of event. The server starts processing it and when there is a blocking IO operation, it does not wait until it completes and instead registers a callback function. The server then immediately starts to process another event ( maybe another request ). When the IO operation is finished, that is another kind of event, and the server will process it ( i.e. continue working on the request ) by executing the callback as soon as it has time.

Node.js Platform does not follow Request/Response Multi-Threaded Stateless Model. It follows Single Threaded with Event Loop Model. Node.js Processing model mainly based on Javascript Event based model with Javascript callback mechanism.

[](https://github.com/Mohamed-Hashem/nodejs-interview-questions/blob/master/assets/event-loop.png)

**Single Threaded Event Loop Model Processing Steps:**

* Clients Send request to Web Server.
* Node.js Web Server internally maintains a Limited Thread pool to provide services to the Client Requests.
* Node.js Web Server receives those requests and places them into a Queue. It is known as **Event Queue**.
* Node.js Web Server internally has a Component, known as **Event Loop**. Why it got this name is that it uses indefinite loop to receive requests and process them.
* Event Loop uses Single Thread only. It is main heart of Node.js Platform Processing Model.
* Event Loop checks any Client Request is placed in Event Queue. If no, then wait for incoming requests for indefinitely.
* If yes, then pick up one Client Request from Event Queue
  + Starts process that Client Request
  + If that Client Request Does Not requires any Blocking IO Operations, then process everything, prepare response and send it back to client.
  + If that Client Request requires some Blocking IO Operations like interacting with Database, File System, External Services then it will follow different approach
    - Checks Threads availability from Internal Thread Pool
    - Picks up one Thread and assign this Client Request to that thread.
    - That Thread is responsible for taking that request, process it, perform Blocking IO operations, prepare response and send it back to the Event Loop
    - Event Loop in turn, sends that Response to the respective Client.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are the core modules of Node.js?**

Node.js has a set of core modules that are part of the platform and come with the Node.js installation. These modules can be loaded into the program by using the require function.

**Syntax:**

const module = require('module\_name');

**Example:**

const http = require('http');

http.createServer(function (req, res) {

res.writeHead(200, {'Content-Type': 'text/html'});

res.write('Welcome to Node.js!');

res.end();

}).listen(3000);

The following table lists some of the important core modules in Node.js.

| **Core Module** | **Description** |
| --- | --- |
| assert | provides a set of assertion functions useful for testing |
| console | provides a simple debugging console |
| crypto | provides cryptographic functionality |
| http | http module includes classes, methods and events to create Node.js http server. |
| url | url module includes methods for URL resolution and parsing. |
| querystring | querystring module includes methods to deal with query string. |
| path | path module includes methods to deal with file paths. |
| fs | fs module includes classes, methods, and events to work with file I/O. |
| util | util module includes utility functions useful for programmers. |

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

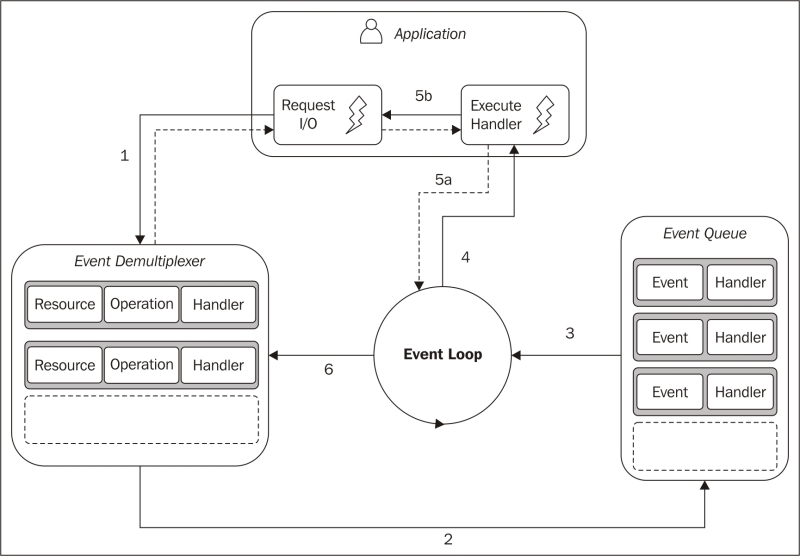
**Q. What do you understand by Reactor Pattern in Node.js?**

**Reactor Pattern** is used to avoid the blocking of the Input/Output operations. It provides us with a handler that is associated with I/O operations. When the I/O requests are to be generated, they get submitted to a demultiplexer, which handles concurrency in avoiding the blocking of the I/O mode and collects the requests in form of an event and queues those events.

**There are two ways in which I/O operations are performed:**

**1. Blocking I/O:** Application will make a function call and pause its execution at a point until the data is received. It is called as "Synchronous".

**2. Non-Blocking I/O:** Application will make a function call, and, without waiting for the results it continues its execution. It is called as "Asynchronous".

[](https://github.com/Mohamed-Hashem/nodejs-interview-questions/blob/master/assets/reactor-pattern.jpg)

**Reactor Pattern comprises of:**

**1. Resources:** They are shared by multiple applications for I/O operations, generally slower in executions.

**2. Synchronous Event De-multiplexer/Event Notifier:** This uses Event Loop for blocking on all resources. When a set of I/O operations completes, the Event De-multiplexer pushes the new events into the Event Queue.

**3. Event Loop and Event Queue:** Event Queue queues up the new events that occurred along with its event-handler, pair.

**4. Request Handler/Application:** This is, generally, the application that provides the handler to be executed for registered events on resources.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are the global objects of node.js?**

Node.js Global Objects are the objects that are available in all modules. Global Objects are built-in objects that are part of the JavaScript and can be used directly in the application without importing any particular module.

These objects are modules, functions, strings and object itself as explained below.

**1. global:**

It is a global namespace. Defining a variable within this namespace makes it globally accessible.

var myvar;

**2. process:**

It is an inbuilt global object that is an instance of EventEmitter used to get information on current process. It can also be accessed using require() explicitly.

**3. console:**

It is an inbuilt global object used to print to stdout and stderr.

console.log("Hello World"); // Hello World

**4. setTimeout(), clearTimeout(), setInterval(), clearInterval():**

The built-in timer functions are globals

function printHello() {

console.log( "Hello, World!");

}

// Now call above function after 2 seconds

var timeoutObj = setTimeout(printHello, 2000);

**5. \_\_dirname:**

It is a string. It specifies the name of the directory that currently contains the code.

console.log(\_\_dirname);

**6. \_\_filename:**

It specifies the filename of the code being executed. This is the resolved absolute path of this code file. The value inside a module is the path to that module file.

console.log(\_\_filename);

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is chrome v8 engine?**

V8 is a C++ based open-source JavaScript engine developed by Google. It was originally designed for Google Chrome and Chromium-based browsers ( such as Brave ) in 2008, but it was later utilized to create Node.js for server-side coding.

V8 is the JavaScript engine i.e. it parses and executes JavaScript code. The DOM, and the other Web Platform APIs ( they all makeup runtime environment ) are provided by the browser.

V8 is known to be a JavaScript engine because it takes JavaScript code and executes it while browsing in Chrome. It provides a runtime environment for the execution of JavaScript code. The best part is that the JavaScript engine is completely independent of the browser in which it runs.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Why is LIBUV needed in Node JS?**

**libuv** is a C library originally written for Node.js to abstract non-blocking I/O operations. It provide the below features

* It allows the CPU and other resources to be used simultaneously while still performing I/O operations, thereby resulting in efficient use of resources and network.
* It facilitates an event-driven approach wherein I/O and other activities are performed using callback-based notifications.
* It provides mechanisms to handle file system, DNS, network, child processes, pipes, signal handling, polling and streaming
* It also includes a thread pool for offloading work for some things that can't be done asynchronously at the operating system level.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How V8 compiles JavaScript code?**

Compilation is the process of converting human-readable code to machine code. There are two ways to compile the code

* **Using an Interpreter**: The interpreter scans the code line by line and converts it into byte code.
* **Using a Compiler**: The Compiler scans the entire document and compiles it into highly optimized byte code.

The V8 engine uses both a compiler and an interpreter and follows **just-in-time (JIT)** compilation to speed up the execution. JIT compiling works by compiling small portions of code that are just about to be executed. This prevents long compilation time and the code being compiles is only that which is highly likely to run.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 5. NODE.JS EVENTS**

**Q. What is EventEmitter in Node.js?**

The EventEmitter is a class that facilitates communication/interaction between objects in Node.js. The EventEmitter class can be used to create and handle custom events.

EventEmitter is at the core of Node asynchronous event-driven architecture. Many of Node's built-in modules inherit from EventEmitter including prominent frameworks like Express.js. An emitter object basically has two main features:

* Emitting name events.
* Registering and unregistering listener functions.

**Example 01:** Create an event emitter instance and register a couple of callbacks

/\*\*

\* Callbacks Events

\*/

const events = require('events');

const eventEmitter = new events.EventEmitter();

function listenerOne() {

console.log('First Listener Executed');

}

function listenerTwo() {

console.log('Second Listener Executed');

}

eventEmitter.on('listenerOne', listenerOne); // Register for listenerOne

eventEmitter.on('listenerOne', listenerTwo); // Register for listenerOne

// When the event "listenerOne" is emitted, both the above callbacks should be invoked.

eventEmitter.emit('listenerOne');

// Output

First Listener Executed

Second Listener Executed

**Example 02:** Registering for the event to be fired only one time using **once**.

/\*\*

\* Emit Events Once

\*/

const events = require('events');

const eventEmitter = new events.EventEmitter();

function listenerOnce() {

console.log('listenerOnce fired once');

}

eventEmitter.once('listenerOne', listenerOnce); // Register listenerOnce

eventEmitter.emit('listenerOne');

// Output

listenerOnce fired once

**Example 03:** Registering for the event with callback parameters

/\*\*

\* Callback Events with Parameters

\*/

const events = require('events');

const eventEmitter = new events.EventEmitter();

function listener(code, msg) {

console.log(`status ${code} and ${msg}`);

}

eventEmitter.on('status', listener); // Register listener

eventEmitter.emit('status', 200, 'ok');

// Output

status 200 and ok

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

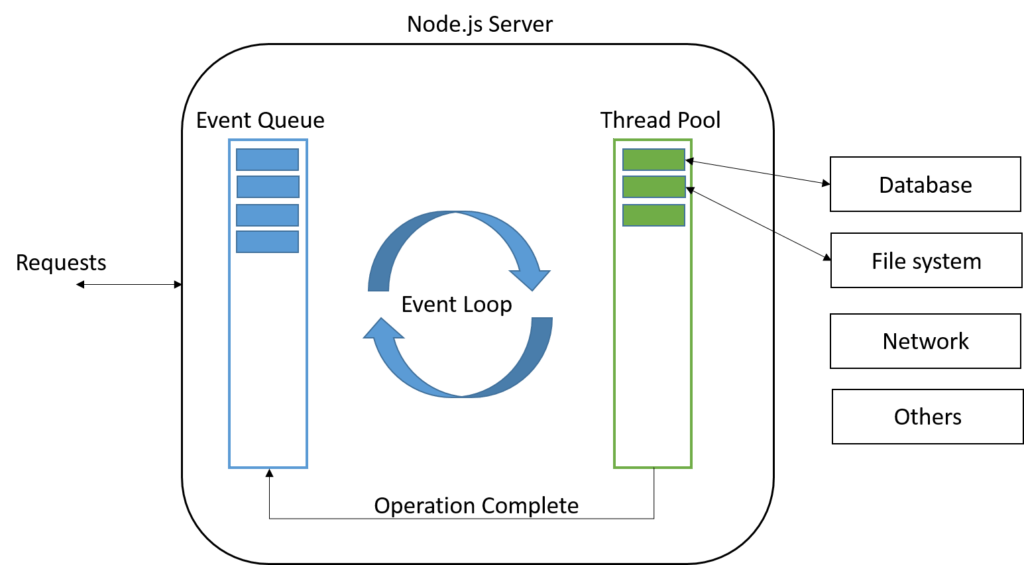
**Q. How the Event Loop Works in Node.js?**

The **event loop** allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded. It is done by offloading operations to the system kernel whenever possible.

Node.js is a single-threaded application, but it can support **concurrency** via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

**Features of Event Loop:**

* Event loop is an endless loop, which waits for tasks, executes them and then sleeps until it receives more tasks.
* The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
* The event loop allows us to use callbacks and promises.
* The event loop executes the tasks starting from the oldest first.

[](https://github.com/Mohamed-Hashem/nodejs-interview-questions/blob/master/assets/nodejs-event-loop.png)

**Example:**

/\*\*

\* Event loop in Node.js

\*/

const events = require('events');

const eventEmitter = new events.EventEmitter();

// Create an event handler as follows

const connectHandler = function connected() {

console.log('connection succesful.');

eventEmitter.emit('data\_received');

}

// Bind the connection event with the handler

eventEmitter.on('connection', connectHandler);

// Bind the data\_received event with the anonymous function

eventEmitter.on('data\_received', function() {

console.log('data received succesfully.');

});

// Fire the connection event

eventEmitter.emit('connection');

console.log("Program Ended.");

// Output

Connection succesful.

Data received succesfully.

Program Ended.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is the difference between process.nextTick() and setImmediate()?**

**1. process.nextTick():**

The process.nextTick() method adds the callback function to the start of the next event queue. It is to be noted that, at the start of the program process.nextTick() method is called for the first time before the event loop is processed.

**2. setImmdeiate():**

The setImmediate() method is used to execute a function right after the current event loop finishes. It is callback function is placed in the check phase of the next event queue.

**Example:**

/\*\*

\* setImmediate() and process.nextTick()

\*/

setImmediate(() => {

console.log("1st Immediate");

});

setImmediate(() => {

console.log("2nd Immediate");

});

process.nextTick(() => {

console.log("1st Process");

});

process.nextTick(() => {

console.log("2nd Process");

});

// First event queue ends here

console.log("Program Started");

// Output

Program Started

1st Process

2nd Process

1st Immediate

2nd Immediate

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is callback function in Node.js?**

A callback is a function which is called when a task is completed, thus helps in preventing any kind of blocking and a callback function allows other code to run in the meantime.

Callback is called when task get completed and is asynchronous equivalent for a function. Using Callback concept, Node.js can process a large number of requests without waiting for any function to return the result which makes Node.js highly scalable.

**Example:**

/\*\*

\* Callback Function

\*/

function myAsync(a, b, callback) {

setTimeout(function () {

callback(a + b);

}, 100);

}

console.log("Before Asynchronous Call");

myAsync(10, 20, function (result) {

console.log("Sum: " + result);

});

console.log("After Asynchronous Call");

// Output

Before Asynchronous Call

After Asynchronous Call

Sum: 30

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are the difference between Events and Callbacks?**

**1. Events:**

Node.js **events** module which emits named events that can cause corresponding functions or callbacks to be called. Functions ( callbacks ) listen or subscribe to a particular event to occur and when that event triggers, all the callbacks subscribed to that event are fired one by one in order to which they were registered.

All objects that emit events are instances of the **EventEmitter** class. The event can be emitted or listen to an event with the help of EventEmitter

**Example:**

/\*\*

\* Events Module

\*/

const event = require('events');

const eventEmitter = new event.EventEmitter();

// add listener function for Sum event

eventEmitter.on('Sum', function(num1, num2) {

console.log('Total: ' + (num1 + num2));

});

// call event

eventEmitter.emit('Sum', 10, 20);

// Output

Total: 30

**2. Callbacks:**

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

**Example:**

/\*\*

\* Callbacks

\*/

function sum(number) {

console.log('Total: ' + number);

}

function calculator(num1, num2, callback) {

let total = num1 + num2;

callback(total);

}

calculator(10, 20, sum);

// Output

Total: 30

Callback functions are called when an asynchronous function returns its result, whereas event handling works on the **observer pattern**. The functions that listen to events act as Observers. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is an error-first callback?**

The pattern used across all the asynchronous methods in Node.js is called *Error-first Callback*. Here is an example:

fs.readFile( "file.json", function ( err, data ) {

if ( err ) {

console.error( err );

}

console.log( data );

});

Any asynchronous method expects one of the arguments to be a callback. The full callback argument list depends on the caller method, but the first argument is always an error object or null. When we go for the asynchronous method, an exception thrown during function execution cannot be detected in a try/catch statement. The event happens after the JavaScript engine leaves the try block.

In the preceding example, if any exception is thrown during the reading of the file, it lands on the callback function as the first and mandatory parameter.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is callback hell in Node.js?**

The callback hell contains complex nested callbacks. Here, every callback takes an argument that is a result of the previous callbacks. In this way, the code structure looks like a pyramid, making it difficult to read and maintain. Also, if there is an error in one function, then all other functions get affected.

An asynchronous function is one where some external activity must complete before a result can be processed; it is "asynchronous" in the sense that there is an unpredictable amount of time before a result becomes available. Such functions require a callback function to handle errors and process the result.

**Example:**

/\*\*

\* Callback Hell

\*/

getData(function(a){

getMoreData(a, function(b){

getMoreData(b, function(c){

getMoreData(c, function(d){

getMoreData(d, function(e){

...

});

});

});

});

});

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to avoid callback hell in Node.js?**

**1. Managing callbacks using Async.js:**

Async is a really powerful npm module for managing asynchronous nature of JavaScript. Along with Node.js, it also works for JavaScript written for browsers.

Async provides lots of powerful utilities to work with asynchronous processes under different scenarios.

npm install --save async

**2. Managing callbacks hell using promises:**

Promises are alternative to callbacks while dealing with asynchronous code. Promises return the value of the result or an error exception. The core of the promises is the .then() function, which waits for the promise object to be returned.

The .then() function takes two optional functions as arguments and depending on the state of the promise only one will ever be called. The first function is called when the promise if fulfilled (A successful result). The second function is called when the promise is rejected.

**Example:**

/\*\*

\* Promises

\*/

const myPromise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Successful!");

}, 300);

});

**3. Using Async Await:**

Async await makes asynchronous code look like it's synchronous. This has only been possible because of the reintroduction of promises into node.js. Async-Await only works with functions that return a promise.

**Example:**

/\*\*

\* Async Await

\*/

const getrandomnumber = function(){

return new Promise((resolve, reject)=>{

setTimeout(() => {

resolve(Math.floor(Math.random() \* 20));

}, 1000);

});

}

const addRandomNumber = async function(){

const sum = await getrandomnumber() + await getrandomnumber();

console.log(sum);

}

addRandomNumber();

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is typically the first argument passed to a callback handler?**

The first parameter of the callback is the **error** value. If the function hits an error, then they typically call the **callback** with the first parameter being an Error object.

**Example:**

/\*\*

\* Callback Handler

\*/

const Division = (numerator, denominator, callback) => {

if (denominator === 0) {

callback(new Error('Divide by zero error!'));

} else {

callback(null, numerator / denominator);

}

};

// Function Call

Division(5, 0, (err, result) => {

if (err) {

return console.log(err.message);

}

console.log(`Result: ${result}`);

});

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are the timing features of Node.js?**

The Timers module in Node.js contains functions that execute code after a set period of time. Timers do not need to be imported via require(), since all the methods are available globally to emulate the browser JavaScript API.

Some of the functions provided in this module are

**1. setTimeout():**

This function schedules code execution after the assigned amount of time ( in milliseconds ). Only after the timeout has occurred, the code will be executed. This method returns an ID that can be used in **clearTimeout()** method.

**Syntax:**

setTimeout(callback, delay, args )

**Example:**

function printMessage(arg) {

console.log(`${arg}`);

}

setTimeout(printMessage, 1000, 'Display this Message after 1 seconds!');

**2. setImmediate():**

The setImmediate() method executes the code at the end of the current event loop cycle. The function passed in the setImmediate() argument is a function that will be executed in the next iteration of the event loop.

**Syntax:**

setImmediate(callback, args)

**Example:**

// Setting timeout for the function

setTimeout(function () {

console.log('setTimeout() function running...');

}, 500);

// Running this function immediately before any other

setImmediate(function () {

console.log('setImmediate() function running...');

});

// Directly printing the statement

console.log('Normal statement in the event loop');

// Output

// Normal statement in the event loop

// setImmediate() function running...

// setTimeout() function running...

**3. setInterval():**

The setInterval() method executes the code after the specified interval. The function is executed multiple times after the interval has passed. The function will keep on calling until the process is stopped externally or using code after specified time period. The clearInterval() method can be used to prevent the function from running.

**Syntax:**

setInterval(callback, delay, args)

**Example:**

setInterval(function() {

console.log('Display this Message intervals of 1 seconds!');

}, 1000);

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 6. NODE.JS FILE SYSTEM**

**Q. How Node.js read the content of a file?**

The "normal" way in Node.js is probably to read in the content of a file in a non-blocking, asynchronous way. That is, to tell Node to read in the file, and then to get a callback when the file-reading has been finished. That would allow us to hand several requests in parallel.

Common use for the File System module:

* Read files
* Create files
* Update files
* Delete files
* Rename files

**Read Files**  
index.html

<html>

<body>

<h1>My Header</h1>

<p>My paragraph.</p>

</body>

</html>

/\*\*

\* read\_file.js

\*/

const http = require('http');

const fs = require('fs');

http.createServer(function (req, res) {

fs.readFile('index.html', function(err, data) {

res.writeHead(200, {'Content-Type': 'text/html'});

res.write(data);

res.end();

});

}).listen(8080);

Initiate read\_file.js:

node read\_file.js

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 7. NODE.JS STREAMS**

**Q. How many types of streams are present in node.js?**

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. There are four types of 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.

**Example:**

* **data** − This event is fired when there is data is available to read.
* **end** − This event is fired when there is no more data to read.
* **error** − This event is fired when there is any error receiving or writing data.
* **finish** − This event is fired when all the data has been flushed to underlying system.

**Reading from a Stream:**

const fs = require("fs");

const data = '';

// Create a readable stream

const readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.

readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error

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

data += chunk;

});

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

console.log(data);

});

readerStream.on('error', function(err) {

console.log(err.stack);

});

console.log("Program Ended");

**Writing to a Stream:**

const fs = require("fs");

const data = 'Simply Easy Learning';

// Create a writable stream

const writerStream = fs.createWriteStream('output.txt');

// Write the data to stream with encoding to be utf8

writerStream.write(data,'UTF8');

// Mark the end of file

writerStream.end();

// Handle stream events --> finish, and error

writerStream.on('finish', function() {

console.log("Write completed.");

});

writerStream.on('error', function(err) {

console.log(err.stack);

});

console.log("Program Ended");

**Piping the Streams:**

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream. There is no limit on piping operations.

const fs = require("fs");

// Create a readable stream

const readerStream = fs.createReadStream('input.txt');

// Create a writable stream

const writerStream = fs.createWriteStream('output.txt');

// Pipe the read and write operations

// read input.txt and write data to output.txt

readerStream.pipe(writerStream);

console.log("Program Ended");

**Chaining the Streams:**

Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations.

const fs = require("fs");

const zlib = require('zlib');

// Compress the file input.txt to input.txt.gz

fs.createReadStream('input.txt')

.pipe(zlib.createGzip())

.pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 8. NODE.JS MULTITHREADING**

**Q. Is Node.js entirely based on a single-thread?**

Yes, it is true that Node.js processes all requests on a single thread. But it is just a part of the theory behind Node.js design. In fact, more than the single thread mechanism, it makes use of events and callbacks to handle a large no. of requests asynchronously.

Moreover, Node.js has an optimized design which utilizes both JavaScript and C++ to guarantee maximum performance. JavaScript executes at the server-side by Google Chrome v8 engine. And the C++ lib UV library takes care of the non-sequential I/O via background workers.

To explain it practically, let's assume there are 100s of requests lined up in Node.js queue. As per design, the main thread of Node.js event loop will receive all of them and forwards to background workers for execution. Once the workers finish processing requests, the registered callbacks get notified on event loop thread to pass the result back to the user.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How does Node.js handle child threads?**

Node.js is a single threaded language which in background uses multiple threads to execute asynchronous code. Node.js is non-blocking which means that all functions ( callbacks ) are delegated to the event loop and they are ( or can be ) executed by different threads. That is handled by Node.js run-time.

* Nodejs Primary application runs in an event loop, which is in a single thread.
* Background I/O is running in a thread pool that is only accessible to C/C++ or other compiled/native modules and mostly transparent to the JS.
* Node v11/12 now has experimental worker\_threads, which is another option.
* Node.js does support forking multiple processes ( which are executed on different cores ).
* It is important to know that state is not shared between master and forked process.
* We can pass messages to forked process ( which is different script ) and to master process from forked process with function send.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How does Node.js support multi-processor platforms, and does it fully utilize all processor resources?**

Since Node.js is by default a single thread application, it will run on a single processor core and will not take full advantage of multiple core resources. However, Node.js provides support for deployment on multiple-core systems, to take greater advantage of the hardware. The Cluster module is one of the core Node.js modules and it allows running multiple Node.js worker processes that will share the same port.

The cluster module helps to spawn new processes on the operating system. Each process works independently, so you cannot use shared state between child processes. Each process communicates with the main process by IPC and pass server handles back and forth.

Cluster supports two types of load distribution:

* The main process listens on a port, accepts new connection and assigns it to a child process in a round robin fashion.
* The main process assigns the port to a child process and child process itself listen the port.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Since node is a single threaded process, how to make use of all CPUs?**

Node.js is a single threaded language which in background uses multiple threads to execute asynchronous code. Node.js is non-blocking which means that all functions ( callbacks ) are delegated to the event loop and they are ( or can be ) executed by different threads. That is handled by Node.js run-time.

* Node.js does support forking multiple processes ( which are executed on different cores ).
* It is important to know that state is not shared between master and forked process.
* We can pass messages to forked process ( which is different script ) and to master process from forked process with function send.

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.

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`);

}

Running Node.js will now share port 8000 between the workers:

$ node server.js

Master 3596 is running

Worker 4324 started

Worker 4520 started

Worker 6056 started

Worker 5644 started

The worker processes are spawned using the child\_process.fork() method, so that they can communicate with the parent via IPC and pass server handles back and forth.

The cluster module supports two methods of distributing incoming connections.

The first one (and the default one on all platforms except Windows), is the round-robin approach, where the master process listens on a port, accepts new connections and distributes them across the workers in a round-robin fashion, with some built-in smarts to avoid overloading a worker process.

The second approach is where the master process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. If Node.js is single threaded then how it handles concurrency?**

Node js despite being single-threaded is the asynchronous nature that makes it possible to handle concurrency and perform multiple I/O operations at the same time. Node js uses an event loop to maintain concurrency and perform non-blocking I/O operations.

As soon as Node js starts, it initializes an event loop. The event loop works on a queue (which is called an event queue) and performs tasks in FIFO (First In First Out) order. It executes a task only when there is no ongoing task in the call stack. The call stack works in LIFO(Last In First Out) order. The event loop continuously checks the call stack to check if there is any task that needs to be run. Now whenever the event loop finds any function, it adds it to the stack and runs in order.

**Example:**

/\*\*

\* Concurrency

\*/

function add(a, b) {

return a + b;

}

function print(n) {

console.log(`Two times the number ${n} is ` + add(n, n));

}

print(5);

Here, the function **print(5)** will be invoked and will push into the call stack. When the function is called, it starts consoling the statement inside it but before consoling the whole statement it encounters another function add(n,n) and suspends its current execution, and pushes the add function into the top of the call stack.

Now the function will return the addition **a+b** and then popped out from the stack and now the previously suspended function will start running and will log the output to console and then this function too will get pop from the stack and now the stack is empty. So this is how a call stack works.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to kill child processes that spawn their own child processes in Node.js?**

If a child process in Node.js spawn their own child processes, kill() method will not kill the child process's own child processes. For example, if I start a process that starts it's own child processes via child\_process module, killing that child process will not make my program to quit.

const spawn = require('child\_process').spawn;

const child = spawn('my-command');

child.kill();

The program above will not quit if my-command spins up some more processes.

**PID range hack:**

We can start child processes with {detached: true} option so those processes will not be attached to main process but they will go to a new group of processes. Then using process.kill(-pid) method on main process we can kill all processes that are in the same group of a child process with the same pid group. In my case, I only have one processes in this group.

const spawn = require('child\_process').spawn;

const child = spawn('my-command', {detached: true});

process.kill(-child.pid);

Please note - before pid. This converts a pid to a group of pids for process kill() method.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to synchronize data between multiple clients on node.js server?**

*ToDo*

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 9. NODE.JS WEB MODULE**

**Q. How to use JSON Web Token (JWT) for authentication in Node.js?**

JSON Web Token (JWT) is an open standard that defines a compact and self-contained way of securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

There are some advantages of using JWT for authorization:

* Purely stateless. No additional server or infra required to store session information.
* It can be easily shared among services.

**Syntax:**

jwt.sign(payload, secretOrPrivateKey, [options, callback])

* **Header** - Consists of two parts: the type of token (i.e., JWT) and the signing algorithm (i.e., HS512)
* **Payload** - Contains the claims that provide information about a user who has been authenticated along with other information such as token expiration time.
* **Signature** - Final part of a token that wraps in the encoded header and payload, along with the algorithm and a secret

**Installation:**

npm install jsonwebtoken bcryptjs --save

**Example**:

/\*\*

\* AuthController.js

\*/

const express = require('express');

const router = express.Router();

const bodyParser = require('body-parser');

const User = require('../user/User');

const jwt = require('jsonwebtoken');

const bcrypt = require('bcryptjs');

const config = require('../config');

router.use(bodyParser.urlencoded({ extended: false }));

router.use(bodyParser.json());

router.post('/register', function(req, res) {

let hashedPassword = bcrypt.hashSync(req.body.password, 8);

User.create({

name : req.body.name,

email : req.body.email,

password : hashedPassword

},

function (err, user) {

if (err) return res.status(500).send("There was a problem registering the user.")

// create a token

let token = jwt.sign({ id: user.\_id }, config.secret, {

expiresIn: 86400 // expires in 24 hours

});

res.status(200).send({ auth: true, token: token });

});

});

**config.js:**

/\*\*

\* config.js

\*/

module.exports = {

'secret': 'supersecret'

};

The jwt.sign() method takes a payload and the secret key defined in config.js as parameters. It creates a unique string of characters representing the payload. In our case, the payload is an object containing only the id of the user.

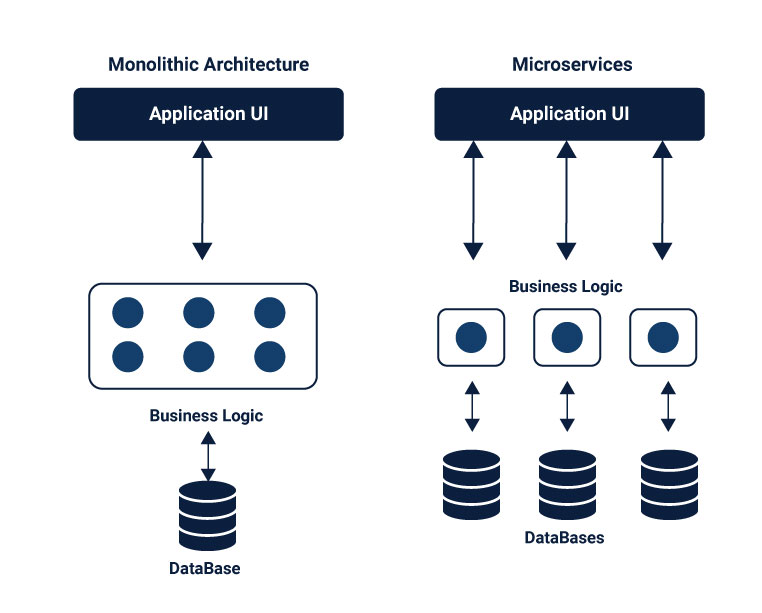
**Reference:**

* [*https://www.npmjs.com/package/jsonwebtoken*](https://www.npmjs.com/package/jsonwebtoken)

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to build a microservices architecture with Node.js?**

Microservices are a style of **Service Oriented Architecture (SOA)** where the app is structured on an assembly of interconnected services. With microservices, the application architecture is built with lightweight protocols. The services are finely seeded in the architecture. Microservices disintegrate the app into smaller services and enable improved modularity.

[](https://github.com/Mohamed-Hashem/nodejs-interview-questions/blob/master/assets/monolithic-and-microservices-architecture.jpg)

There are few things worth emphasizing about the superiority of microservices, and distributed systems generally, over monolithic architecture:

* **Modularity** — responsibility for specific operations is assigned to separate pieces of the application
* **Uniformity** — microservices interfaces (API endpoints) consist of a base URI identifying a data object and standard HTTP methods (GET, POST, PUT, PATCH and DELETE) used to manipulate the object
* **Robustness** — component failures cause only the absence or reduction of a specific unit of functionality
* **Maintainability** — system components can be modified and deployed independently
* **Scalability** — instances of a service can be added or removed to respond to changes in demand.
* **Availability** — new features can be added to the system while maintaining 100% availability.
* **Testability** — new solutions can be tested directly in the production environment by implementing them for restricted segments of users to see how they behave in real life.

**Example:** Creating Microservices with Node.js

**Step 01:** Creating a Server to Accept Requests

This file is creating our server and assigns routes to process all requests.

// server.js

const express = require('express')

const app = express();

const port = process.env.PORT || 3000;

const routes = require('./api/routes');

routes(app);

app.listen(port, function() {

console.log('Server started on port: ' + port);

});

**Step 02:** Defining the routes

The next step is to define the routes for the microservices and then assign each to a target in the controller. We have two endpoints. One endpoint called "about" that returns information about the application. And a "distance" endpoint that includes two path parameters, both Zip Codes of the Lego store. This endpoint returns the distance, in miles, between these two Zip Codes.

const controller = require('./controller');

module.exports = function(app) {

app.route('/about')

.get(controller.about);

app.route('/distance/:zipcode1/:zipcode2')

.get(controller.getDistance);

};

**Step 03:** Adding Controller Logic

Within the controller file, we are going to create a controller object with two properties. Those properties are the functions to handle the requests we defined in the routes module.

const properties = require('../package.json')

const distance = require('../service/distance');

const controllers = {

about: function(req, res) {

let aboutInfo = {

name: properties.name,

version: properties.version

}

res.json(aboutInfo);

},

getDistance: function(req, res) {

distance.find(req, res, function(err, dist) {

if (err)

res.send(err);

res.json(dist);

});

},

};

module.exports = controllers;

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How microservices communicate with each other in Node.js?**

*ToDo*

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 10. NODE.JS MIDDLEWARE**

**Q. What are the middleware functions in Node.js?**

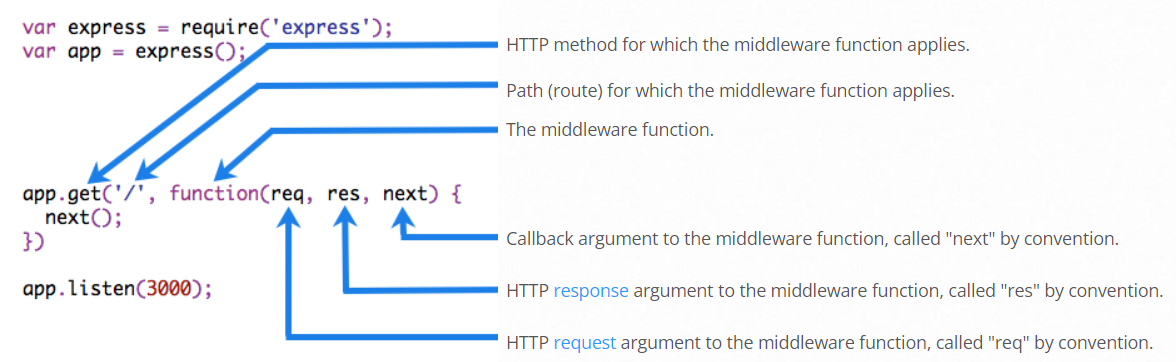
Middleware functions are functions that have access to the **request object (req)**, the **response object (res)**, and the next function in the application's request-response cycle.

Middleware functions can perform the following tasks:

* Execute any code.
* Make changes to the request and the response objects.
* End the request-response cycle.
* Call the next middleware in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

The following figure shows the elements of a middleware function call:

[](https://github.com/Mohamed-Hashem/nodejs-interview-questions/blob/master/assets/express-mw.png)

Middleware functions that return a Promise will call next(value) when they reject or throw an error. next will be called with either the rejected value or the thrown Error.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain the use of next in Node.js?**

The **next** is a function in the Express router which executes the middleware succeeding the current middleware.

**Example:**

To load the middleware function, call app.use(), specifying the middleware function. For example, the following code loads the **myLogger** middleware function before the route to the root path (/).

/\*\*

\* myLogger

\*/

const express = require("express");

const app = express();

const myLogger = function (req, res, next) {

console.log("LOGGED");

next();

};

app.use(myLogger);

app.get("/", (req, res) => {

res.send("Hello World!");

});

app.listen(3000);

**⚝** [**Try this example on CodeSandbox**](https://codesandbox.io/s/next-function-nq042s)

*Note: The next() function is not a part of the Node.js or Express API, but is the third argument that is passed to the middleware function. The next() function could be named anything, but by convention it is always named “next”. To avoid confusion, always use this convention.*

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Why to use Express.js?**

Express.js is a Node.js web application framework that provides broad features for building web and mobile applications. It is used to build a single page, multipage, and hybrid web application.

**Features of Express.js:**

* **Fast Server-Side Development:** The features of node js help express saving a lot of time.
* **Middleware:** Middleware is a request handler that has access to the application's request-response cycle.
* **Routing:** It refers to how an application's endpoint's URLs respond to client requests.
* **Templating:** It provides templating engines to build dynamic content on the web pages by creating HTML templates on the server.
* **Debugging:** Express makes it easier as it identifies the exact part where bugs are.

The Express.js framework makes it very easy to develop an application which can be used to handle multiple types of requests like the GET, PUT, and POST and DELETE requests.

**Example:**

/\*\*

\* Simple server using Express.js

\*/

const express = require("express");

const app = express();

app.get("/", function (req, res) {

res.send("Hello World!");

});

const server = app.listen(3000, function () {});

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Why should you separate Express 'app' and 'server'?**

Keeping the API declaration separated from the network related configuration (port, protocol, etc) allows testing the API in-process, without performing network calls, with all the benefits that it brings to the table: fast testing execution and getting coverage metrics of the code. It also allows deploying the same API under flexible and different network conditions.

API declaration, should reside in app.js:

/\*\*

\* app.js

\*/

const app = express();

app.use(bodyParser.json());

app.use("/api/events", events.API);

app.use("/api/forms", forms);

Server network declaration

/\*\*

\* server.js

\*/

const app = require('../app');

const http = require('http');

// Get port from environment and store in Express.

const port = normalizePort(process.env.PORT || '3000');

app.set('port', port);

// Create HTTP server.

const server = http.createServer(app);

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are some of the most popular packages of Node.js?**

| **Package** | **Description** |
| --- | --- |
| Async | Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript |
| Browserify | Browserify will recursively analyze all the require() calls in your app in order to build a bundle you can serve up to the browser in a single <script> tag |
| Bower | Bower is a package manager for the web It works by fetching and installing packages from all over, taking care of hunting, finding, downloading, and saving the stuff you're looking for |
| Csv | csv module has four sub modules which provides CSV generation, parsing, transformation and serialization for Node.js |
| Debug | Debug is a tiny node.js debugging utility modelled after node core's debugging technique |
| Express | Express is a fast, un-opinionated, minimalist web framework. It provides small, robust tooling for HTTP servers, making it a great solution for single page applications, web sites, hybrids, or public HTTP APIs |
| Grunt | is a JavaScript Task Runner that facilitates creating new projects and makes performing repetitive but necessary tasks such as linting, unit testing, concatenating and minifying files (among other things) trivial |
| Gulp | is a streaming build system that helps you automate painful or time-consuming tasks in your development workflow |
| Hapi | is a streaming build system that helps you automate painful or time-consuming tasks in your development workflow |
| Http-server | is a simple, zero-configuration command-line http server. It is powerful enough for production usage, but it's simple and hackable enough to be used for testing, local development, and learning |
| Inquirer | A collection of common interactive command line user interfaces |
| Jquery | jQuery is a fast, small, and feature-rich JavaScript library |
| Jshint | Static analysis tool to detect errors and potential problems in JavaScript code and to enforce your team's coding conventions |
| Koa | Koa is web app framework. It is an expressive HTTP middleware for node.js to make web applications and APIs more enjoyable to write |
| Lodash | The lodash library exported as a node module. Lodash is a modern JavaScript utility library delivering modularity, performance, & extras |
| Less | The less library exported as a node module |
| Moment | A lightweight JavaScript date library for parsing, validating, manipulating, and formatting dates |
| Mongoose | It is a MongoDB object modeling tool designed to work in an asynchronous environment |
| MongoDB | The official MongoDB driver for Node.js. It provides a high-level API on top of mongodb-core that is meant for end users |
| Npm | is package manager for javascript |
| Nodemon | It is a simple monitor script for use during development of a node.js app, It will watch the files in the directory in which nodemon was started, and if any files change, nodemon will automatically restart your node application |
| Nodemailer | This module enables e-mail sending from a Node.js applications |
| Optimist | is a node.js library for option parsing with an argv hash |
| Phantomjs | An NPM installer for PhantomJS, headless webkit with JS API. It has fast and native support for various web standards: DOM handling, CSS selector, JSON, Canvas, and SVG |
| Passport | A simple, unobtrusive authentication middleware for Node.js. Passport uses the strategies to authenticate requests. Strategies can range from verifying username and password credentials or authentication using OAuth or OpenID |
| Q | Q is a library for promises. A promise is an object that represents the return value or the thrown exception that the function may eventually provide |
| Request | Request is Simplified HTTP request client make it possible to make http calls. It supports HTTPS and follows redirects by default |
| Socket.io | Its a node.js realtime framework server |
| Sails | Sails is a API-driven framework for building realtime apps, using MVC conventions (based on Express and Socket.io) |
| Through | It enables simplified stream construction. It is easy way to create a stream that is both readable and writable |
| Underscore | Underscore.js is a utility-belt library for JavaScript that provides support for the usual functional suspects (each, map, reduce, filter...) without extending any core JavaScript objects |
| Validator | A nodejs module for a library of string validators and sanitizers |
| Winston | A multi-transport async logging library for Node.js |
| Ws | A simple to use, blazing fast and thoroughly tested websocket client, server and console for node.js |
| Xml2js | A Simple XML to JavaScript object converter |
| Yo | A CLI tool for running Yeoman generators |
| Zmq | Bindings for node.js and io.js to ZeroMQ. It is a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications |

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How can you make sure your dependencies are safe?**

The only option is to automate the update / security audit of your dependencies. For that there are free and paid options:

1. npm outdated
2. Trace by RisingStack
3. NSP
4. GreenKeeper
5. Snyk
6. npm audit
7. npm audit fix

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are the security mechanisms available in Node.js?**

**1. Helmet module:**

[Helmet](https://www.npmjs.com/package/helmet) helps to secure your Express applications by setting various HTTP headers, like:

* X-Frame-Options to mitigates clickjacking attacks,
* Strict-Transport-Security to keep your users on HTTPS,
* X-XSS-Protection to prevent reflected XSS attacks,
* X-DNS-Prefetch-Control to disable browsers DNS prefetching.

/\*\*

\* Helmet

\*/

const express = require('express')

const helmet = require('helmet')

const app = express()

app.use(helmet())

**2. JOI module:**

Validating user input is one of the most important things to do when it comes to the security of your application. Failing to do it correctly can open up your application and users to a wide range of attacks, including command injection, SQL injection or stored cross-site scripting.

To validate user input, one of the best libraries you can pick is joi. [Joi](https://www.npmjs.com/package/joi) is an object schema description language and validator for JavaScript objects.

/\*\*

\* Joi

\*/

const Joi = require('joi');

const schema = Joi.object().keys({

username: Joi.string().alphanum().min(3).max(30).required(),

password: Joi.string().regex(/^[a-zA-Z0-9]{3,30}$/),

access\_token: [Joi.string(), Joi.number()],

birthyear: Joi.number().integer().min(1900).max(2013),

email: Joi.string().email()

}).with('username', 'birthyear').without('password', 'access\_token')

// Return result

const result = Joi.validate({

username: 'abc',

birthyear: 1994

}, schema)

// result.error === null -> valid

**3. Regular Expressions:**

Regular Expressions are a great way to manipulate texts and get the parts that you need from them. However, there is an attack vector called Regular Expression Denial of Service attack, which exposes the fact that most Regular Expression implementations may reach extreme situations for specially crafted input, that cause them to work extremely slowly.

The Regular Expressions that can do such a thing are commonly referred as Evil Regexes. These expressions contain: \*grouping with repetition, \*inside the repeated group: \*repetition, or \*alternation with overlapping

Examples of Evil Regular Expressions patterns:

(a+)+

([a-zA-Z]+)\*

(a|aa)+

**4. Security.txt:**

Security.txt defines a standard to help organizations define the process for security researchers to securely disclose security vulnerabilities.

const express = require('express')

const securityTxt = require('express-security.txt')

const app = express()

app.get('/security.txt', securityTxt({

// your security address

contact: 'email@example.com',

// your pgp key

encryption: 'encryption',

// if you have a hall of fame for securty resourcers, include the link here

acknowledgements: 'http://acknowledgements.example.com'

}))

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is npm in Node.js?**

NPM stands for Node Package Manager. It provides following two main functionalities.

* It works as an Online repository for node.js packages/modules which are present at <nodejs.org>.
* It works as Command line utility to install packages, do version management and dependency management of Node.js packages. NPM comes bundled along with Node.js installable. We can verify its version using the following command-

npm --version

NPM helps to install any Node.js module using the following command.

npm install <Module Name>

For example, following is the command to install a famous Node.js web framework module called express-

npm install express

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Why npm shrinkwrap is useful?**

NPM shrinkwrap lets you lock down the ver­sions of installed pack­ages and their descen­dant pack­ages. It helps you use same package versions on all environments (development, staging, production) and also improve download and installation speed.

After installing packages using npm install or npm install <package-name> and updating your **node\_modules** folder, you should run

npm shrinkwrap

It should create new **npm-shrinkwrap.json** file with information about all packages you use. Next time, when someone calls **npm install**, it will install packages from **npm-shrinkwrap.json** and you will have the same environment on all machines.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to handle file upload in Node.js?**

File can be uploaded to the server using Multer module. Multer is a Node.js middleware which is used for handling multipart/form-data, which is mostly used library for uploading files.

**1. Installing the dependencies:**

npm install express body-parser multer --save

**2. server.js:**

/\*\*

\* File Upload in Node.js

\*/

const express = require("express");

const bodyParser = require("body-parser");

const multer = require("multer");

const app = express();

// for text/number data transfer between clientg and server

app.use(bodyParser());

const storage = multer.diskStorage({

destination: function (req, file, callback) {

callback(null, "./uploads");

},

filename: function (req, file, callback) {

callback(null, file.fieldname + "-" + Date.now());

},

});

const upload = multer({ storage: storage }).single("userPhoto");

app.get("/", function (req, res) {

res.sendFile(\_\_dirname + "/index.html");

});

// POST: upload for single file upload

app.post("/api/photo", function (req, res) {

upload(req, res, function (err) {

if (err) {

return res.end("Error uploading file.");

}

res.end("File is uploaded");

});

});

app.listen(3000, function () {

console.log("Listening on port 3000");

});

**3. index.html:**

<!DOCTYPE html>

<html lang="en">

<head>

<title>Multer-File-Upload</title>

</head>

<body>

<h1>MULTER File Upload | Single File Upload</h1>

<form id = "uploadForm"

enctype = "multipart/form-data"

action = "/api/photo"

method = "post"

>

<input type="file" name="userPhoto" />

<input type="submit" value="Upload Image" name="submit">

</form>

</body>

</html>

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain the terms body-parser, cookie-parser, morgan, nodemon, pm2, serve-favicon, cors, dotenv, fs-extra, moment in Express.js?**

**1. body-parser:**

body-parser extract the entire body portion of an incoming request stream and exposes it on req.body. The body-parser module parses the JSON, buffer, string and URL encoded data submitted using HTTP POST request.

**Example:**

npm install body-parser

/\*\*

\* body-parser

\*/

const express = require("express");

const bodyParser = require("body-parser");

const app = express();

// create application/json parser

const jsonParser = bodyParser.json();

// create application/x-www-form-urlencoded parser

const urlencodedParser = bodyParser.urlencoded({ extended: false });

// POST /login gets urlencoded bodies

app.post("/login", urlencodedParser, function (req, res) {

res.send("welcome, " + req.body.username);

});

// POST /api/users gets JSON bodies

app.post("/api/users", jsonParser, function (req, res) {

// create user in req.body

});

**2. cookie-parser:**

A cookie is a piece of data that is sent to the client-side with a request and is stored on the client-side itself by the Web Browser the user is currently using.

The cookie-parser middleware's cookieParser function takes a secret string or array of strings as the first argument and an options object as the second argument.

**Installation:**

npm install cookie-parser

**Example:**

/\*\*

\* cookie-parser

\*/

const express = require('express')

const cookieParser = require('cookie-parser')

const app = express()

app.use(cookieParser())

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

// Cookies that have not been signed

console.log('Cookies: ', req.cookies)

// Cookies that have been signed

console.log('Signed Cookies: ', req.signedCookies)

})

app.listen(3000)

**3. morgan:**

HTTP request logger middleware for node.js.

**Installation:**

npm install morgan

**Example:**

/\*\*

\* Writing logs to a file

\*/

const express = require('express')

const fs = require('fs')

const morgan = require('morgan')

const path = require('path')

const app = express()

// create a write stream (in append mode)

const accessLogStream = fs.createWriteStream(path.join(\_\_dirname, 'access.log'), { flags: 'a' })

// setup the logger

app.use(morgan('combined', { stream: accessLogStream }))

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

res.send('hello, world!')

})

**4. nodemon:**

Nodemon is a utility that will monitor for any changes in source and automatically restart your server.

**Installation:**

npm install -g nodemon

**Example:**

{

// ...

"scripts": {

"start": "nodemon server.js"

},

// ...

}

**5. pm2:**

**P**(rocess) **M**(anager) **2** (pm2) is a production process manager for Node.js applications with a built-in load balancer. It allows to keep applications alive forever, to reload them without downtime and to facilitate common system admin tasks.

**Installation:**

npm install pm2 -g

**Start an application:**

pm2 start app.js

**Reference:**

* [*https://pm2.keymetrics.io/docs/usage/quick-start/*](https://pm2.keymetrics.io/docs/usage/quick-start/)

**6. serve-favicon:**

Node.js middleware for serving a favicon. It create new middleware to serve a favicon from the given path to a favicon file. **path** may also be a Buffer of the icon to serve.

**Installation:**

npm install serve-favicon

**Example:**

/\*\*

\* serve-favicon

\*/

const express = require('express')

const favicon = require('serve-favicon')

const path = require('path')

const app = express()

app.use(favicon(path.join(\_\_dirname, 'public', 'favicon.ico')))

// Add your routes here, etc.

app.listen(3000)

**7. cors:**

**C**ross-**O**rigin **R**esource **S**haring (CORS) headers allow apps running in the browser to make requests to servers on different domains (also known as origins). CORS headers are set on the server side - the HTTP server is responsible for indicating that a given HTTP request can be cross-origin.

**Installation:**

npm install cors

**Example:**

/\*\*

\* Enable CORS for a Single Route

\*/

const express = require('express')

const cors = require('cors')

const app = express()

app.get('/products/:id', cors(), function (req, res, next) {

res.json({msg: 'This is CORS-enabled for a Single Route'})

})

app.listen(8080, function () {

console.log('CORS-enabled web server listening on port 80')

})

**8. dotenv:**

When a NodeJs application runs, it injects a global variable called process.env which contains information about the state of environment in which the application is running. The dotenv loads environment variables stored in the .env file into process.env.

**Installation:**

npm install dotenv

**Usage:**

// .env

DB\_HOST=localhost

DB\_USER=admin

DB\_PASS=root

/\*\*

\* config.js

\*/

const db = require('db')

db.connect({

host: process.env.DB\_HOST,

username: process.env.DB\_USER,

password: process.env.DB\_PASS

})

**9. fs-extra:**

fs-extra contains methods that aren't included in the vanilla Node.js fs package. Such as recursive mkdir, copy, and remove. It also uses graceful-fs to prevent EMFILE errors.

**Installation:**

npm install fs-extra

**Usage:**

/\*\*

\* fs-extra

\*/

const fs = require('fs-extra')

// Async with callbacks:

fs.copy('/tmp/myfile', '/tmp/mynewfile', err => {

if (err) return console.error(err)

console.log('success!')

})

**10. moment:**

A JavaScript date library for parsing, validating, manipulating, and formatting dates.

**Installation:**

npm install moment --save

**Usage:**

* Format Dates

const moment = require('moment');

moment().format('MMMM Do YYYY, h:mm:ss a'); // October 24th 2022, 3:15:22 pm

moment().format('dddd'); // Saturday

moment().format("MMM Do YY"); // Oct 24th 22

* Relative Time

const moment = require('moment');

moment("20111031", "YYYYMMDD").fromNow(); // 9 years ago

moment("20120620", "YYYYMMDD").fromNow(); // 8 years ago

moment().startOf('day').fromNow(); // 15 hours ago

* Calendar Time

const moment = require('moment');

moment().subtract(10, 'days').calendar(); // 10/14/2022

moment().subtract(6, 'days').calendar(); // Last Sunday at 3:18 PM

moment().subtract(3, 'days').calendar(); // Last Wednesday at 3:18 PM

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 11. NODE.JS RESTFUL API**

**Q. Explain RESTful Web Services in Node.js?**

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It is an architectural style as well as an approach for communications purposes that is often used in various web services development. A REST Server simply provides access to resources and REST client accesses and modifies the resources using HTTP protocol.

**HTTP methods:**

* GET − Provides read-only access to a resource.
* PUT − Updates an existing resource or creates a new resource.
* DELETE − Removes a resource.
* POST − Creates a new resource.
* PATCH− Update/modify a resource

**Example:**

// users.json

{

"user1" : {

"id": 1,

"name" : "Ehsan Philip",

"age" : 24

},

"user2" : {

"id": 2,

"name" : "Karim Jimenez",

"age" : 22

},

"user3" : {

"id": 3,

"name" : "Giacomo Weir",

"age" : 18

}

}

**List Users** ( GET method)

Let's implement our first RESTful API listUsers using the following code in a server.js file −

const express = require('express');

const app = express();

const fs = require("fs");

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

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

console.log( data );

res.end( data );

});

})

const server = app.listen(3000, function () {

const host = server.address().address

const port = server.address().port

console.log("App listening at http://%s:%s", host, port)

});

**Add User** ( POST method )

Following API will show you how to add new user in the list.

const express = require('express');

const app = express();

const fs = require("fs");

const user = {

"user4" : {

"id": 4,

"name" : "Spencer Amos",

"age" : 28

}

}

app.post('/addUser', function (req, res) {

// First read existing users.

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

data = JSON.parse( data );

data["user4"] = user["user4"];

console.log( data );

res.end( JSON.stringify(data));

});

})

const server = app.listen(3000, function () {

const host = server.address().address

const port = server.address().port

console.log("App listening at http://%s:%s", host, port)

})

**Delete User:**

const express = require('express');

const app = express();

const fs = require("fs");

const id = 2;

app.delete('/deleteUser', function (req, res) {

// First read existing users.

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

data = JSON.parse( data );

delete data["user" + 2];

console.log( data );

res.end( JSON.stringify(data));

});

})

const server = app.listen(3000, function () {

const host = server.address().address

const port = server.address().port

console.log("App listening at http://%s:%s", host, port)

})

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is the difference between req.params and req.query?**

The **req.params** are a part of a path in URL and they're also known as URL variables. for example, if you have the route **/books/:id**, then the **id** property will be available as **req.params.id**. req.params default value is an empty object {}.

A **req.query** is a part of a URL that assigns values to specified parameters. A query string commonly includes fields added to a base URL by a Web browser or other client application, for example as part of an HTML form. A query is the last part of URL

**Example 01:** req.params

/\*\*

\* req.params

\*/

// GET http://localhost:3000/employees/10

app.get('/employees/:id', (req, res, next) => {

console.log(req.params.id); // 10

})

**Example 02:** req.query

/\*\*

\* req.query

\*/

// GET http://localhost:3000/employees?page=20

app.get('/employees', (req, res, next) => {

console.log(req.query.page) // 20

})

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to make post request in Node.js?**

Following code snippet can be used to make a Post Request in Node.js.

/\*\*

\* POST Request

\*/

const request = require("request");

request.post("http://localhost:3000/action", { form: { key: "value" } },

function (error, response, body) {

if (!error && response.statusCode === 200) {

console.log(body);

}

}

);

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are Promises in Node.js?**

It allows to associate handlers to an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of the final value, the asynchronous method returns a promise for the value at some point in the future.

Promises in node.js promised to do some work and then had separate callbacks that would be executed for success and failure as well as handling timeouts. Another way to think of promises in node.js was that they were emitters that could emit only two events: success and error.The cool thing about promises is you can combine them into dependency chains (do Promise C only when Promise A and Promise B complete).

The core idea behind promises is that a promise represents the result of an asynchronous operation. A promise is in one of three different states:

* pending - The initial state of a promise.
* fulfilled - The state of a promise representing a successful operation.
* rejected - The state of a promise representing a failed operation. Once a promise is fulfilled or rejected, it is immutable (i.e. it can never change again).

**Creating a Promise:**

const myPromise = new Promise(function(resolve, reject){

....

})

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How can you secure your HTTP cookies against XSS attacks?**

**1.** When the web server sets cookies, it can provide some additional attributes to make sure the cookies won't be accessible by using malicious JavaScript. One such attribute is HttpOnly.

Set-Cookie: [name]=[value]; HttpOnly

HttpOnly makes sure the cookies will be submitted only to the domain they originated from.

**2.** The "Secure" attribute can make sure the cookies are sent over secured channel only.

Set-Cookie: [name]=[value]; Secure

**3.** The web server can use X-XSS-Protection response header to make sure pages do not load when they detect reflected cross-site scripting (XSS) attacks.

X-XSS-Protection: 1; mode=block

**4.** The web server can use HTTP Content-Security-Policy response header to control what resources a user agent is allowed to load for a certain page. It can help to prevent various types of attacks like Cross Site Scripting (XSS) and data injection attacks.

Content-Security-Policy: default-src 'self' \*.http://sometrustedwebsite.com

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to make an HTTP POST request using Node.js?**

const https = require('https')

const obj = {

"userId":1,

"id":1,

"title":"whatever",

"completed":false

}

const data = JSON.stringify(obj)

const options = {

hostname: 'jsonplaceholder.typicode.com',

port: 443,

path: '/todos',

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Content-Length': data.length

}

}

const req = https.request(options, res => {

console.log(`statusCode: ${res.statusCode}`)

res.on('data', d => {

process.stdout.write(d)

})

})

req.on('error', error => {

console.error(error)

})

req.write(data)

req.end()

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is asynchronous programming in Node.js?**

Asynchronous programming is a form of parallel programming that allows a unit of work to run separately from the primary application thread. When the work is complete, it notifies the main thread (as well as whether the work was completed or failed). There are numerous benefits to using it, such as improved application performance and enhanced responsiveness.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is the difference between Asynchronous and Non-blocking?**

**1. Asynchronous:**

The architecture of asynchronous explains that the message sent will not give the reply on immediate basis just like we send the mail but do not get the reply on an immediate basis. It does not have any dependency or order. Hence improving the system efficiency and performance. The server stores the information and when the action is done it will be notified.

**2. Non-Blocking:**

Nonblocking immediately responses with whatever data available. Moreover, it does not block any execution and keeps on running as per the requests. If an answer could not be retrieved then in those cases API returns immediately with an error. Nonblocking is mostly used with I/O(input/output). Node.js is itself based on nonblocking I/O model. There are few ways of communication that a nonblocking I/O has completed. The callback function is to be called when the operation is completed. Nonblocking call uses the help of javascript which provides a callback function.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How node.js prevents blocking code?**

**Blocking vs Non-blocking:**

**Blocking** is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a **blocking** operation is occurring.

Synchronous methods in the Node.js standard library that use **libuv** are the most commonly used blocking operations. Native modules may also have blocking methods. Blocking methods execute synchronously and non-blocking methods execute asynchronously.

**Example:**

// Blocking

const fs = require('fs');

const data = fs.readFileSync('/file.md'); // blocks here until file is read

console.log(data);

moreWork(); // will run after console.log

// Non-blocking

const fs = require('fs');

fs.readFile('/file.md', (err, data) => {

if (err) throw err;

console.log(data);

});

moreWork(); // will run before console.log

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Name the types of API functions in Node.js?**

There are two types of API functions in Node.js:

* Asynchronous, Non-blocking functions
* Synchronous, Blocking functions

**1. Blocking functions:**

In a blocking operation, all other code is blocked from executing until an I/O event that is being waited on occurs. Blocking functions execute synchronously.

**Example:**

const fs = require('fs');

const data = fs.readFileSync('/file.md'); // blocks here until file is read

console.log(data);

// moreWork(); will run after console.log

The second line of code blocks the execution of additional JavaScript until the entire file is read. moreWork () will only be called after Console.log

**2. Non-blocking functions:**

In a non-blocking operation, multiple I/O calls can be performed without the execution of the program being halted. Non-blocking functions execute asynchronously.

**Example:**

const fs = require('fs');

fs.readFile('/file.md', (err, data) => {

if (err) throw err;

console.log(data);

});

// moreWork(); will run before console.log

Since fs.readFile() is non-blocking, moreWork() does not have to wait for the file read to complete before being called. This allows for higher throughput.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is difference between put and patch?**

PUT and PATCH are HTTP verbs and they both relate to updating a resource. The main difference between PUT and PATCH requests are in the way the server processes the enclosed entity to modify the resource identified by the Request-URI.

In a PUT request, the enclosed entity is considered to be a modified version of the resource stored on the origin server, and the client is requesting that the stored version be replaced.

With PATCH, however, the enclosed entity contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version.

Also, another difference is that when you want to update a resource with PUT request, you have to send the full payload as the request whereas with PATCH, you only send the parameters which you want to update.

The most commonly used HTTP verbs POST, GET, PUT, DELETE are similar to CRUD (Create, Read, Update and Delete) operations in database. We specify these HTTP verbs in the capital case. So, the below is the comparison between them.

* POST - create
* GET - read
* PUT - update
* DELETE - delete

**PATCH**: Submits a partial modification to a resource. If you only need to update one field for the resource, you may want to use the PATCH method.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. List types of Http requests supported by Node.js?**

The HTTP core module is a key module to Node.js networking.

const http = require('http')

**http.METHODS**

require('http').METHODS

[ 'ACL',

'BIND',

'CHECKOUT',

'CONNECT',

'COPY',

'DELETE',

'GET',

'HEAD',

'LINK',

'LOCK',

'M-SEARCH',

'MERGE',

'MKACTIVITY',

'MKCALENDAR',

'MKCOL',

'MOVE',

'NOTIFY',

'OPTIONS',

'PATCH',

'POST',

'PROPFIND',

'PROPPATCH',

'PURGE',

'PUT',

'REBIND',

'REPORT',

'SEARCH',

'SUBSCRIBE',

'TRACE',

'UNBIND',

'UNLINK',

'UNLOCK',

'UNSUBSCRIBE' ]

**http.STATUS\_CODES**

require('http').STATUS\_CODES

{ '100': 'Continue',

'101': 'Switching Protocols',

'102': 'Processing',

'200': 'OK',

'201': 'Created',

'202': 'Accepted',

'203': 'Non-Authoritative Information',

'204': 'No Content',

'205': 'Reset Content',

'206': 'Partial Content',

'207': 'Multi-Status',

'208': 'Already Reported',

'226': 'IM Used',

'300': 'Multiple Choices',

'301': 'Moved Permanently',

'302': 'Found',

'303': 'See Other',

'304': 'Not Modified',

'305': 'Use Proxy',

'307': 'Temporary Redirect',

'308': 'Permanent Redirect',

'400': 'Bad Request',

'401': 'Unauthorized',

'402': 'Payment Required',

'403': 'Forbidden',

'404': 'Not Found',

'405': 'Method Not Allowed',

'406': 'Not Acceptable',

'407': 'Proxy Authentication Required',

'408': 'Request Timeout',

'409': 'Conflict',

'410': 'Gone',

'411': 'Length Required',

'412': 'Precondition Failed',

'413': 'Payload Too Large',

'414': 'URI Too Long',

'415': 'Unsupported Media Type',

'416': 'Range Not Satisfiable',

'417': 'Expectation Failed',

'418': 'I\'m a teapot',

'421': 'Misdirected Request',

'422': 'Unprocessable Entity',

'423': 'Locked',

'424': 'Failed Dependency',

'425': 'Unordered Collection',

'426': 'Upgrade Required',

'428': 'Precondition Required',

'429': 'Too Many Requests',

'431': 'Request Header Fields Too Large',

'451': 'Unavailable For Legal Reasons',

'500': 'Internal Server Error',

'501': 'Not Implemented',

'502': 'Bad Gateway',

'503': 'Service Unavailable',

'504': 'Gateway Timeout',

'505': 'HTTP Version Not Supported',

'506': 'Variant Also Negotiates',

'507': 'Insufficient Storage',

'508': 'Loop Detected',

'509': 'Bandwidth Limit Exceeded',

'510': 'Not Extended',

'511': 'Network Authentication Required' }

**Making HTTP Requests**

const request = require('request');

request('https://nodejs.org/', function(err, res, body) {

console.log(body);

});

The first argument to request can either be a URL string, or an object of options. Here are some of the more common options you'll encounter in your applications:

* **url**: The destination URL of the HTTP request
* **method**: The HTTP method to be used (GET, POST, DELETE, etc)
* **headers**: An object of HTTP headers (key-value) to be set in the request
* **form**: An object containing key-value form data

const request = require('request');

const options = {

url: 'https://nodejs.org/file.json',

method: 'GET',

headers: {

'Accept': 'application/json',

'Accept-Charset': 'utf-8',

'User-Agent': 'my-reddit-client'

}

};

request(options, function(err, res, body) {

let json = JSON.parse(body);

console.log(json);

});

Using the options object, this request uses the GET method to retrieve JSON data directly from Reddit, which is returned as a string in the body field. From here, you can use JSON.parse and use the data as a normal JavaScript object.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is difference between promises and async-await in Node.js?**

**1. Promises:**

A promise is used to handle the asynchronous result of an operation. JavaScript is designed to not wait for an asynchronous block of code to completely execute before other synchronous parts of the code can run. With Promises, we can defer the execution of a code block until an async request is completed. This way, other operations can keep running without interruption.

**States of Promises:**

* Pending: Initial State, before the Promise succeeds or fails.
* Resolved: Completed Promise
* Rejected: Failed Promise, throw an error

**Example:**

function logFetch(url) {

return fetch(url)

.then(response => {

console.log(response);

})

.catch(err => {

console.error('fetch failed', err);

});

}

**2. Async-Await:**

Await is basically syntactic sugar for **Promises**. It makes asynchronous code look more like synchronous/procedural code, which is easier for humans to understand.

Putting the keyword async before a function tells the function to return a Promise. If the code returns something that is not a Promise, then JavaScript automatically wraps it into a resolved promise with that value. The await keyword simply makes JavaScript wait until that Promise settles and then returns its result.

**Example:**

async function logFetch(url) {

try {

const response = await fetch(url);

console.log(response);

}

catch (err) {

console.log('fetch failed', err);

}

}

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Mention the steps by which you can async in Node.js?**

ES 2017 introduced Asynchronous functions. Async functions are essentially a cleaner way to work with asynchronous code in JavaScript.

**1. Async/Await:**

* The newest way to write asynchronous code in JavaScript.
* It is non blocking (just like promises and callbacks).
* Async/Await was created to simplify the process of working with and writing chained promises.
* Async functions return a Promise. If the function throws an error, the Promise will be rejected. If the function returns a value, the Promise will be resolved.

Syntax

// Normal Function

function add(x,y){

return x + y;

}

// Async Function

async function add(x,y){

return x + y;

}

**2. Await:**

Async functions can make use of the await expression. This will pause the async function and wait for the Promise to resolve prior to moving on.

**Example:**

function doubleAfter2Seconds(x) {

return new Promise(resolve => {

setTimeout(() => {

resolve(x \* 2);

}, 2000);

});

}

async function addAsync(x) {

const a = await doubleAfter2Seconds(10);

const b = await doubleAfter2Seconds(20);

const c = await doubleAfter2Seconds(30);

return x + a + b + c;

}

addAsync(10).then((sum) => {

console.log(sum);

});

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to use Q promise in Node.js?**

A promise is an object that represents the return value or the thrown exception that the function may eventually provide. A promise can also be used as a proxy for a remote object to overcome latency.

Promise is relatively an easy implementation for asynchronous operation. The promise object returned from the function represents an operation which is not completed yet, but it guarantees to the caller of the operation that the operation will be completed in future.

Promise has the following states:

* **Pending** - asynchronous operation is not yet completed.
* **Fulfilled** - asynchronous operation is completed successfully.
* **Rejected** - asynchronous operation is terminated with an error.
* **Settled** - asynchronous operation is either fulfilled or rejected.
* **Callback** - function is executed if the promise is executed with value.
* **Errback** - function is executed if the promise is rejected.

**Moving to Promises from Callback**

On the first pass, promises can mitigate the **Pyramid of Doom**: the situation where code marches to the right faster than it marches forward.

step1(function (value1) {

step2(value1, function(value2) {

step3(value2, function(value3) {

step4(value3, function(value4) {

// Do something with value4

});

});

});

});

With a promise library, it can flatten the pyramid.

Q.fcall(promisedStep1)

.then(promisedStep2)

.then(promisedStep3)

.then(promisedStep4)

.then(function (value4) {

// Do something with value4

})

.catch(function (error) {

// Handle any error from all above steps

})

.done();

**Reference:**

* [*https://www.npmjs.com/package/q*](https://www.npmjs.com/package/q)

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What are async functions in Node?**

**Q. How do you convert an existing callback API to promises?**

*ToDo*

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 12. NODE.JS ROUTING**

**Q. How does routing work in Node.js?**

Routing defines the way in which the client requests are handled by the application endpoints. We define routing using methods of the Express app object that correspond to HTTP methods; for example, app.get() to handle GET requests and app.post to handle POST requests, app.all() to handle all HTTP methods and app.use() to specify middleware as the callback function.

These routing methods "listens" for requests that match the specified route(s) and method(s), and when it detects a match, it calls the specified callback function.

*Syntax*:

app.METHOD(PATH, HANDLER)

Where:

* app is an instance of express.
* METHOD is an HTTP request method.
* PATH is a path on the server.
* HANDLER is the function executed when the route is matched.

**a) Route methods:**

// GET method route

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

res.send('GET request')

})

// POST method route

app.post('/login', function (req, res) {

res.send('POST request')

})

// ALL method route

app.all('/secret', function (req, res, next) {

console.log('Accessing the secret section ...')

next() // pass control to the next handler

})

**b) Route paths:**

Route paths, in combination with a request method, define the endpoints at which requests can be made. Route paths can be strings, string patterns, or regular expressions.

The characters ?, +, \*, and () are subsets of their regular expression counterparts. The hyphen (-) and the dot (.) are interpreted literally by string-based paths.

**Example:**

// This route path will match requests to /about.

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

res.send('about')

})

// This route path will match acd and abcd.

app.get('/ab?cd', function (req, res) {

res.send('ab?cd')

})

// This route path will match butterfly and dragonfly

app.get(/.\*fly$/, function (req, res) {

res.send('/.\*fly$/')

})

**c) Route parameters:**

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the req.params object, with the name of the route parameter specified in the path as their respective keys.

**Example:**

app.get('/users/:userId', function (req, res) {

res.send(req.params)

})

**Response methods:**

| **Method** | **Description** |
| --- | --- |
| res.download() | Prompt a file to be downloaded. |
| res.end() | End the response process. |
| res.json() | Send a JSON response. |
| res.jsonp() | Send a JSON response with JSONP support. |
| res.redirect() | Redirect a request. |
| res.render() | Render a view template. |
| res.send() | Send a response of various types. |
| res.sendFile() | Send a file as an octet stream. |
| res.sendStatus() | Set the response status code and send its string representation as the response body. |

**d) Router method:**

const express = require('express')

const router = express.Router()

// middleware that is specific to this router

router.use(function timeLog (req, res, next) {

console.log('Time: ', Date.now())

next()

})

// define the home page route

router.get('/', function (req, res) {

res.send('Birds home page')

})

// define the about route

router.get('/about', function (req, res) {

res.send('About birds')

})

module.exports = router

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 13. NODE.JS CACHING**

**# 14. NODE.JS ERROR HANDLING**

**Q. What is the preferred method of resolving unhandled exceptions in Node.js?**

Unhandled exceptions in Node.js can be caught at the Process level by attaching a handler for uncaughtException event.

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

console.log('Caught exception: ' + err);

});

Process is a global object that provides information about the current Node.js process. Process is a listener function that is always listening to events.

Few events are :

1. Exit
2. disconnect
3. unhandledException
4. rejectionHandled

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain Error Handling approaches in Node.js?**

**Q. How would you handle errors for async code in Node.js?**

**Q. How to solve "Process out of Memory Exception" in Node.js?**

**Q. What are the types of memory leaks in node.js**

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 15. NODE.JS LOGGING**

**Q. How to debug an application in Node.js?**

**1. node-inspector:**

npm install -g node-inspector

Run

node-debug app.js

**2. Debugging:**

* Debugger
* Node Inspector
* Visual Studio Code
* Cloud9
* Brackets

**3. Profiling:**

1. node --prof ./app.js

2. node --prof-process ./the-generated-log-file

**4. Heapdumps:**

* node-heapdump with Chrome Developer Tools

**5. Tracing:**

* Interactive Stack Traces with TraceGL

**6. Logging:**

Libraries that output debugging information

* Caterpillar
* Tracer
* scribbles

Libraries that enhance stack trace information

* Longjohn

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 16. NODE.JS INTERNATIONALIZATION**

**Q. How to use locale (i18n) in Node.js?**

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 17. NODE.JS TESTING**

**Q. What is a stub?**

Stubbing and verification for node.js tests. Enables you to validate and override behaviour of nested pieces of code such as methods, require() and npm modules or even instances of classes. This library is inspired on node-gently, MockJS and mock-require.

**Features of Stub:**

* Produces simple, lightweight Objects capable of extending down their tree
* Compatible with Nodejs
* Easily extendable directly or through an ExtensionManager
* Comes with predefined, usable extensions

Stubs are functions/programs that simulate the behaviours of components/modules. Stubs provide canned answers to function calls made during test cases. Also, you can assert on with what these stubs were called.

A use-case can be a file read, when you do not want to read an actual file:

const fs = require('fs');

const readFileStub = sinon.stub(fs, 'readFile', function (path, cb) {

return cb(null, 'filecontent');

});

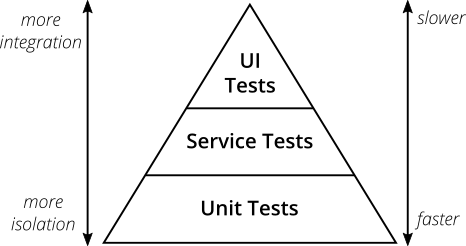
expect(readFileStub).to.be.called;

readFileStub.restore();

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is a test pyramid?**

The "Test Pyramid" is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these groups. It shows which kinds of tests you should be looking for in the different levels of the pyramid and gives practical examples on how these can be implemented.

[](https://github.com/Mohamed-Hashem/nodejs-interview-questions/blob/master/assets/testPyramid.png)

Mike Cohn's original test pyramid consists of three layers that your test suite should consist of (bottom to top):

1. Unit Tests
2. Service Tests
3. User Interface Tests

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to use Joi module for schema validation in Node.js?**

Joi module is a popular module for data validation. This module validates the data based on schemas. There are various functions like optional(), required(), min(), max(), etc which make it easy to use and a user-friendly module for validating the data.

**Example:**

const Joi = require("joi");

// User-defined function to validate the user

function validateUser(user) {

const JoiSchema = Joi.object({

username: Joi.string().min(5).max(30).required(),

email: Joi.string().email().min(5).max(50).optional(),

date\_of\_birth: Joi.date().optional(),

account\_status: Joi.string()

.valid("activated")

.valid("unactivated")

.optional(),

}).options({ abortEarly: false });

return JoiSchema.validate(user);

}

const user = {

username: "Deepak Lucky",

email: "deepak.lucky@gmail.com",

date\_of\_birth: "2000-07-07",

account\_status: "activated",

};

let response = validateUser(user);

if (response.error) {

console.log(response.error.details);

} else {

console.log("Validated Data");

}

**⚝** [**Try this example on CodeSandbox**](https://codesandbox.io/s/schema-validation-using-joi-s2nhzs)

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**# 18. NODE.JS MISCELLANEOUS**

**Q. What is crypto in Node.js?**

The Node.js Crypto module supports cryptography. It provides cryptographic functionality that includes a set of wrappers for open SSL's hash HMAC, cipher, decipher, sign and verify functions.

* **Hash**: A hash is a fixed-length string of bits i.e. procedurally and deterministically generated from some arbitrary block of source data.
* **HMAC**: HMAC stands for Hash-based Message Authentication Code. It is a process for applying a hash algorithm to both data and a secret key that results in a single final hash.
* Encryption Example using Hash and HMAC

const crypto = require('crypto');

const secret = 'abcdefg';

const hash = crypto.createHmac('sha256', secret)

.update('Welcome to Node.js')

.digest('hex');

console.log(hash);

* Encryption example using Cipher

const crypto = require('crypto');

const cipher = crypto.createCipher('aes192', 'a password');

const encrypted = cipher.update('Hello Node.js', 'utf8', 'hex');

encrypted += cipher.final('hex');

console.log(encrypted);

* Decryption example using Decipher

const crypto = require('crypto');

const decipher = crypto.createDecipher('aes192', 'a password');

const encrypted = '4ce3b761d58398aed30d5af898a0656a3174d9c7d7502e781e83cf6b9fb836d5';

const decrypted = decipher.update(encrypted, 'hex', 'utf8');

decrypted += decipher.final('utf8');

console.log(decrypted);

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to execute an external program from within Node.js?**

const { exec } = require('child\_process');

exec('"/path/to/test file/test.sh" arg1 arg2');

exec('echo "The \\$HOME variable is $HOME"');

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is REPL?**

REPL (READ, EVAL, PRINT, LOOP) is a computer environment similar to Shell (Unix/Linux) and command prompt. Node comes with the REPL environment when it is installed. System interacts with the user through outputs of commands/expressions used. It is useful in writing and debugging the codes. The work of REPL can be understood from its full form:

* **Read**: It reads the inputs from users and parses it into JavaScript data structure. It is then stored to memory.
* **Eval**: The parsed JavaScript data structure is evaluated for the results.
* **Print**: The result is printed after the evaluation.
* **Loop**: Loops the input command. To come out of NODE REPL, press ctrl+c twice

Simple Expression

$ node

> 10 + 20

30

> 10 + ( 20 \* 30 ) - 40

570

>

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What does the runtime environment mean in Node.js?**

The Node.js runtime is the software stack responsible for installing your web service's code and its dependencies and running your service.

The Node.js runtime for App Engine in the standard environment is declared in the app.yaml file:

runtime: nodejs10

The runtime environment is literally just the environment your application is running in. This can be used to describe both the hardware and the software that is running your application. How much RAM, what version of node, what operating system, how much CPU cores, can all be referenced when talking about a runtime environment.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. Explain usage of NODE\_ENV?**

NODE\_ENV is an environment variable made popular by the express web server framework. When a node application is run, it can check the value of the environment variable and do different things based on the value.

For example, when we work on a project and there are production and development environments. We don't need to use caching in the development env. So we set

NODE\_ENV=development

and use the code below

if (process.env.NODE\_ENV === 'development')

useCaching = false;

Upon that, if the project runs on production it will use caching.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How assert works in Node.js?**

The assert module provides a way of testing expressions. If the expression evaluates to 0, or false, an assertion failure is being caused, and the program is terminated.

This module was built to be used internally by Node.js.

// Sample usage

const assert = require('assert');

assert(50 > 70, "50 is less than 70.");

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is the use of DNS module in Node.js?**

DNS is a node module used to do name resolution facility which is provided by the operating system as well as used to do an actual DNS lookup. No need for memorising IP addresses – DNS servers provide a nifty solution of converting domain or subdomain names to IP addresses. This module provides an asynchronous network wrapper and can be imported using the following syntax.

const dns = require('dns');

**Example:** dns.lookup() function

const dns = require('dns');

dns.lookup('www.google.com', (err, addresses, family) => {

console.log('addresses:', addresses);

console.log('family:',family);

});

**Example:** resolve4() and reverse() functions

const dns = require('dns');

dns.resolve4('www.google.com', (err, addresses) => {

if (err) throw err;

console.log(`addresses: ${JSON.stringify(addresses)}`);

addresses.forEach((a) => {

dns.reverse(a, (err, hostnames) => {

if (err) {

throw err;

}

console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);

});

});

});

**Example:** Print the localhost name using lookupService() function

const dns = require('dns');

dns.lookupService('127.0.0.1', 22, (err, hostname, service) => {

console.log(hostname, service);

// Prints: localhost

});

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. What is JIT and how is it related to Node.js?**

Node.js has depended on the V8 JavaScript engine to provide code execution in the language. The V8 is a JavaScript engine built at the google development center, in Germany. It is open source and written in C++. It is used for both client side (Google Chrome) and server side (node.js) JavaScript applications. A central piece of the V8 engine that allows it to execute JavaScript at high speed is the JIT (Just In Time) compiler. This is a dynamic compiler that can optimize code during runtime. When V8 was first built the JIT Compiler was dubbed FullCodegen. Then, the V8 team implemented Crankshaft, which included many performance optimizations that FullCodegen did not implement.

The V8 was first designed to increase the performance of the JavaScript execution inside web browsers. In order to obtain speed, V8 translates JavaScript code into more efficient machine code instead of using an interpreter. It compiles JavaScript code into machine code at execution by implementing a JIT (Just-In-Time) compiler like a lot of modern JavaScript engines such as SpiderMonkey or Rhino (Mozilla) are doing. The main difference with V8 is that it doesn't produce bytecode or any intermediate code.

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to access cache data in Node.js ?**

*ToDo*

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to implement caching in Node.js using Redis?**

*ToDo*

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to implement Memcached in Node.js?**

**Memcached** is a general-purpose distributed memory caching system. It is often used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source (such as a database or API) must be read. Memcached is free and open-source software, licensed under the Revised BSD licence. Memcached runs on Unix-like operating systems (at least LINUX and OS X) and on Microsoft windows.

We can store data to memcached server in key pair format. So whenever any request come from the app can be matched with memcached server without any query from mysql/Nosql server. This increases the performance of the application.

**Installation:**

npm install memcached

**Setting up the client:**

The constructor of the memcached client take 2 different arguments server locations and options. Syntax:

const Memcached = require('memcached');

const memcached = new Memcached(Server locations, options);

**Example:**

/\*\*

\* Memcached

\*/

const Memcached = require('memcached');

// all global configurations should be applied to the .config object of the Client.

Memcached.config.poolSize = 25;

const memcached = new Memcached('localhost:11211', { retries:10, retry:10000, remove:true, failOverServers:['192.168.0.103:11211']});

**Reference:**

* [*https://www.npmjs.com/package/memcached*](https://www.npmjs.com/package/memcached)

[**↥ back to top**](https://github.com/Mohamed-Hashem/nodejs-interview-questions#table-of-contents)

**Q. How to generate and verify checksum of the given string in Nodejs**

The **checksum** (aka **hash sum**) calculation is a one-way process of mapping an extensive data set of variable length (e.g., message, file), to a smaller data set of a fixed length (hash). The length depends on a hashing algorithm.

For the checksum generation, we can use node crypto() module. The module uses createHash(algorithm) to create a checksum (hash) generator. The algorithm is dependent on the available algorithms supported by the version of OpenSSL on the platform.

**Example:**

const crypto = require('crypto');

// To get a list of all available hash algorithms

crypto.getHashes() // [ 'md5', 'sha1', 'sha3-256', ... ]

// Create hash of SHA1 type

const key = "MY\_SECRET\_KEY";

// 'digest' is the output of hash function containing

// only hexadecimal digits

hashPwd = crypto.createHash('sha1').update(key).digest('hex');

console.log(hashPwd); //ef5225a03e4f9cc953ab3c4dd41f5c4db7dc2e5b

**Second Author**

**Interview Questions**

**Node.js**

**Q1: What do you mean by Asynchronous API? ☆☆**

**Answer:** All APIs of Node.js library are aynchronous that is non-blocking. It essentially means a Node.js based server never waits for a API to return data. Server moves to next API after calling it and a notification mechanism of Events of Node.js helps server to get response from the previous API call.

**Source:** *tutorialspoint.com*

**Q2: What are the benefits of using Node.js? ☆☆**

**Answer:** Following are main benefits of using Node.js

* **Aynchronous and Event Driven** - All APIs of Node.js library are aynchronous that is non-blocking. It essentially means a Node.js based server never waits for a API to return data. Server moves to next API after calling it and a notification mechanism of Events of Node.js helps server to get response from the previous API call.
* **Very Fast** - Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
* **Single Threaded but highly Scalable** - Node.js uses a single threaded model with event looping. Event mechanism helps server to respond in a non-bloking ways and makes server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and same program can services much larger number of requests than traditional server like Apache HTTP Server.
* **No Buffering** - Node.js applications never buffer any data. These applications simply output the data in chunks.

**Source:** *tutorialspoint.com*

**Q3: Is Node a single threaded application? ☆☆**

**Answer:** Yes! Node uses a single threaded model with event looping.

**Source:** *tutorialspoint.com*

**Q4: What is global installation of dependencies? ☆☆**

**Answer:** Globally installed packages/dependencies are stored in /npm directory. Such dependencies can be used in CLI (Command Line Interface) function of any node.js but can not be imported using require() in Node application directly. To install a Node project globally use -g flag.

**Source:** *tutorialspoint.com*

**Q5: What is an error-first callback? ☆☆**

**Answer:** *Error-first callbacks* are used to pass errors and data. The first argument is always an error object that the programmer has to check if something went wrong. Additional arguments are used to pass data.

fs.readFile(filePath, function(err, data) {

if (err) {

//handle the error

}

// use the data object

});

**Source:** *tutorialspoint.com*

**Q6: What's the difference between operational and programmer errors? ☆☆**

**Answer:** Operation errors are not bugs, but problems with the system, like *request timeout* or *hardware failure*. On the other hand programmer errors are actual bugs.

**Source:** *blog.risingstack.com*

**Q7: What is the difference between Nodejs, AJAX, and jQuery? ☆☆**

**Answer:** The one common trait between Node.js, AJAX, and jQuery is that all of them are the advanced implementation of JavaScript. However, they serve completely different purposes.

* Node.js –It is a server-side platform for developing client-server applications. For example, if we’ve to build an online employee management system, then we won’t do it using client-side JS. But the Node.js can certainly do it as it runs on a server similar to Apache, Django not in a browser.
* AJAX (aka Asynchronous Javascript and XML) –It is a client-side scripting technique, primarily designed for rendering the contents of a page without refreshing it. There are a no. of large companies utilizing AJAX such as Facebook and Stack Overflow to display dynamic content.
* jQuery –It is a famous JavaScript module which complements AJAX, DOM traversal, looping and so on. This library provides many useful functions to help in JavaScript development. However, it’s not mandatory to use it but as it also manages cross-browser compatibility, so can help you produce highly maintainable web applications.

**Source:** *techbeamers.com*

**Q8: How to make Post request in Node.js? ☆☆**

**Answer:** Following code snippet can be used to make a Post Request in Node.js.

var request = require('request');

request.post('http://www.example.com/action', {

form: {

key: 'value'

}

}, function(error, response, body) {

if (!error && response.statusCode == 200) {

console.log(body)

}

});

**Source:** *techbeamers.com*

**Q9: What are the key features of Node.js? ☆☆**

**Answer:** Let’s look at some of the key features of Node.js.

* **Asynchronous event driven IO helps concurrent request handling –** All APIs of Node.js are asynchronous. This feature means that if a Node receives a request for some Input/Output operation, it will execute that operation in the background and continue with the processing of other requests. Thus it will not wait for the response from the previous requests.
* **Fast in Code execution –** Node.js uses the V8 JavaScript Runtime engine, the one which is used by Google Chrome. Node has a wrapper over the JavaScript engine which makes the runtime engine much faster and hence processing of requests within Node.js also become faster.
* **Single Threaded but Highly Scalable –** Node.js uses a single thread model for event looping. The response from these events may or may not reach the server immediately. However, this does not block other operations. Thus making Node.js highly scalable. Traditional servers create limited threads to handle requests while Node.js creates a single thread that provides service to much larger numbers of such requests.
* **Node.js library uses JavaScript –** This is another important aspect of Node.js from the developer’s point of view. The majority of developers are already well-versed in JavaScript. Hence, development in Node.js becomes easier for a developer who knows JavaScript.
* **There is an Active and vibrant community for the Node.js framework –** The active community always keeps the framework updated with the latest trends in the web development.
* **No Buffering –** Node.js applications never buffer any data. They simply output the data in chunks.

**Source:** *techbeamers.com*

**Q10: What is control flow function? ☆☆**

**Answer:** It is a generic piece of code which runs in between several asynchronous function calls is known as control flow function.

**Source:** *lazyquestion.com*

**Q11: What are Event Listeners? ☆☆**

**Answer:** **Event Listeners** are similar to call back functions but are associated with some event. For example when a server listens to http request on a given port a event will be generated and to specify http server has received and will invoke corresponding event listener. Basically, Event listener's are also call backs for a corresponding event.

Node.js has built in event's and built in event listeners. Node.js also provides functionality to create Custom events and Custom Event listeners.

**Source:** *lazyquestion.com*

**Q12: If Node.js is single threaded then how it handles concurrency? ☆☆**

**Answer:** Node provides a single thread to programmers so that code can be written easily and without bottleneck. Node internally uses multiple POSIX threads for various I/O operations such as File, DNS, Network calls etc.

When Node gets I/O request it creates or uses a thread to perform that I/O operation and once the operation is done, it pushes the result to the event queue. On each such event, event loop runs and checks the queue and if the execution stack of Node is empty then it adds the queue result to execution stack.

This is how Node manages concurrency.

**Source:** *codeforgeek.com*

**Q13: What is Callback Hell? ☆☆**

**Answer:** The asynchronous function requires callbacks as a return parameter. When multiple asynchronous functions are chained together then callback hell situation comes up.

**Source:** *codeforgeek.com*

**Q14: Could we run an external process with Node.js? ☆☆**

**Answer:** Yes. *Child process module* enables us to access operating system functionaries or other apps. Scalability is baked into Node and child processes are the key factors to scale our application. You can use child process to run system commands, read large files without blocking event loop, decompose the application into various “nodes” (That’s why it’s called Node).

Child process module has following three major ways to create child processes –

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

**Source:** *codeforgeek.com*

**Q15: List out the differences between AngularJS and NodeJS? ☆☆**

**Answer:** AngularJS is a web application development framework. It’s a JavaScript and it is different from other web app frameworks written in JavaScript like jQuery. NodeJS is a runtime environment used for building server-side applications while AngularJS is a JavaScript framework mainly useful in building/developing client-side part of applications which run inside a web browser.

**Source:** *a4academics.com*

**Q16: How you can monitor a file for modifications in Node.js ? ☆☆**

**Answer:** We can take advantage of File System watch() function which watches the changes of the file.

**Source:** *codingdefined.com*

**Q17: What are the core modules of Node,js? ☆☆**

**Answer:**

* EventEmitter
* Stream
* FS
* Net
* Global Objects

**Source:** *github.com/jimuyouyou*

**Q18: What is V8? ☆☆**

**Answer:** The V8 library provides Node.js with a JavaScript engine (a program that converts Javascript code into lower level or machine code that microprocessors can understand), which Node.js controls via the V8 C++ API. V8 is maintained by Google, for use in Chrome.

The Chrome V8 engine :

* The V8 engine is written in C++ and used in Chrome and Nodejs.
* It implements ECMAScript as specified in ECMA-262.
* The V8 engine can run standalone we can embed it with our own C++ program.

**Source:** *nodejs.org*

**Q19: What is libuv? ☆☆**

**Answer:** **libuv** is a C library that is used to abstract non-blocking I/O operations to a consistent interface across all supported platforms. It provides mechanisms to handle file system, DNS, network, child processes, pipes, signal handling, polling and streaming. It also includes a thread pool for offloading work for some things that can't be done asynchronously at the operating system level.

**Source:** *nodejs.org*

**Q20: What is the difference between returning a callback and just calling a callback? ☆☆**

**Answer:**

return callback();

//some more lines of code; - won't be executed

callback();

//some more lines of code; - will be executed

Of course returning will help the context calling async function get the value returned by callback.

function do2(callback) {

log.trace('Execute function: do2');

return callback('do2 callback param');

}

var do2Result = do2((param) => {

log.trace(`print ${param}`);

return `return from callback(${param})`; // we could use that return

});

log.trace(`print ${do2Result}`);

Output:

C:\Work\Node>node --use-strict main.js

[0] Execute function: do2

[0] print do2 callback param

[0] print return from callback(do2 callback param)

**Source:** *stackoverflow.com*

**Q21: What is REPL in context of Node? ☆☆☆**

**Answer:** **REPL** stands for Read Eval Print Loop and it represents a computer environment like a window console or unix/linux shell where a command is entered and system responds with an output. Node.js or Node comes bundled with a REPL environment. It performs the following desired tasks.

* **Read** - Reads user's input, parse the input into JavaScript data-structure and stores in memory.
* **Eval** - Takes and evaluates the data structure
* **Print** - Prints the result
* **Loop** - Loops the above command until user press ctrl-c twice.

**Source:** *tutorialspoint.com*

**Q22: What is Callback? ☆☆☆**

**Answer:** **Callback** is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All APIs of Node are written is such a way that they supports callbacks.

For example, a function to read a file may start reading file and return the control to execution environment immediately so that next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as parameter. So there is no blocking or wait for File I/O.

This makes Node.js highly scalable, as it can process high number of request without waiting for any function to return result.

**Source:** *tutorialspoint.com*

**Q23: What is a blocking code? ☆☆☆**

**Answer:** If application has to wait for some I/O operation in order to complete its execution any further then the code responsible for waiting is known as blocking code.

**Source:** *tutorialspoint.com*

**Q24: How Node prevents blocking code? ☆☆☆**

**Answer:** By providing callback function. Callback function gets called whenever corresponding event triggered.

**Source:** *tutorialspoint.com*

**Q25: What is Event Loop? ☆☆☆**

**Answer:** Node.js is a single threaded application but it support concurrency via concept of event and callbacks. As every API of Node js are asynchronous and being a single thread, it uses async function calls to maintain the concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever any task get completed, it fires the corresponding event which signals the event listener function to get executed.

**Source:** *tutorialspoint.com*

**Q26: What is Event Emmitter? ☆☆☆**

**Answer:** All objects that emit events are members of EventEmitter class. These objects expose an eventEmitter.on() function that allows one or more functions to be attached to named events emitted by the object.

When the EventEmitter object emits an event, all of the functions attached to that specific event are called synchronously.

const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('event', () => {

console.log('an event occurred!');

});

myEmitter.emit('event');

**Source:** *tutorialspoint.com*

**Q27: What is purpose of Buffer class in Node? ☆☆☆**

**Answer:** **Buffer** class is a global class and can be accessed in application without importing buffer module. A Buffer is a kind of an array of integers and corresponds to a raw memory allocation outside the V8 heap. A Buffer cannot be resized.

**Source:** *tutorialspoint.com*

**Q28: What is difference between synchronous and asynchronous method of fs module? ☆☆☆**

**Answer:**

Every method in fs module has synchronous as well as asynchronous form. Asynchronous methods takes a last parameter as completion function callback and first parameter of the callback function is error. It is preferred to use asynchronous method instead of synchronous method as former never block the program execution where the latter one does.

**Source:** *tutorialspoint.com*

**Q29: What are streams? ☆☆☆**

**Answer:** Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of 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.

**Source:** *tutorialspoint.com*

**Q30: What is Chaining in Node? ☆☆☆**

**Answer:** **Chanining** is a mechanism to connect output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations.

**Source:** *tutorialspoint.com*

**Q31: What is the purpose of setTimeout function? ☆☆☆**

**Answer:** The setTimeout(cb, ms) global function is used to run callback cb after at least ms milliseconds. The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days.

**Source:** *tutorialspoint.com*

**Q32: How can you avoid callback hells? ☆☆☆**

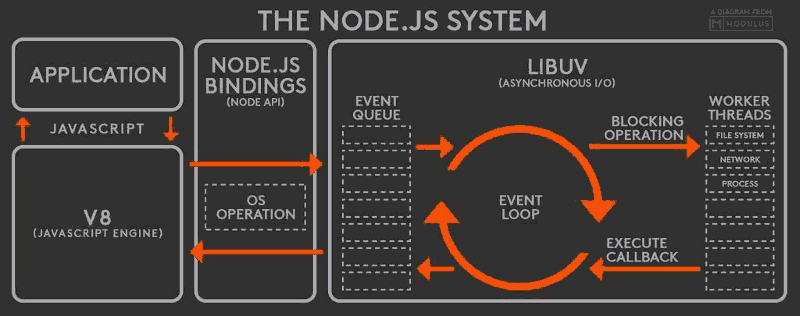
**Answer:** To do so you have more options:

* **modularization**: break callbacks into independent functions
* use *Promises*
* use yield with *Generators* and/or *Promises*

**Source:** *tutorialspoint.com*

**Q33: What's the event loop? ☆☆☆**

**Answer:** **The event loop** is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

[](https://camo.githubusercontent.com/e2bf4a78840ac7af27c6d94078d315323f87bcdfe4593b473aaf319f037e10c1/68747470733a2f2f692e737461636b2e696d6775722e636f6d2f4c6273397a2e706e67)

Every I/O requires a callback - once they are done they are pushed onto the event loop for execution. Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.

**Source:** *blog.risingstack.com*

**Q34: How to avoid callback hell in Node.js? ☆☆☆**

**Answer:** Node.js internally uses a single-threaded event loop to process queued events. But this approach may lead to blocking the entire process if there is a task running longer than expected.

Node.js addresses this problem by incorporating callbacks also known as higher-order functions. So whenever a long-running process finishes its execution, it triggers the callback associated.

sometimes, it could lead to complex and unreadable code. More the no. of callbacks, longer the chain of returning callbacks would be.

There are four solutions which can address the callback hell problem.

**Make your program modular**

It proposes to split the logic into smaller modules. And then join them together from the main module to achieve the desired result.

**Use async mechanism**

It is a widely used Node.js module which provides a sequential flow of execution.

The async module has <async.waterfall> API which passes data from one operation to other using the next callback.

Another async API <async.map> allows iterating over a list of items in parallel and calls back with another list of results.

With the async approach, the caller’s callback gets called only once. The caller here is the main method using the async module.

**Use promises mechanism**

Promises give an alternate way to write async code. They either return the result of execution or the error/exception. Implementing promises requires the use of <.then()> function which waits for the promise object to return. It takes two optional arguments, both functions. Depending on the state of the promise only one of them will get called. The first function call proceeds if the promise gets fulfilled. However, if the promise gets rejected, then the second function will get called.

**Use generators**

Generators are lightweight routines, they make a function wait and resume via the yield keyword. Generator functions uses a special syntax <function\* ()>. They can also suspend and resume asynchronous operations using constructs such as promises or and turn a synchronous code into asynchronous.

**Source:** *techbeamers.com*

**Q35: Explain how does Node.js work? ☆☆☆**

**Answer:** A Node.js application creates a single thread on its invocation. Whenever Node.js receives a request, it first completes its processing before moving on to the next request.

Node.js works asynchronously by using the event loop and callback functions, to handle multiple requests coming in parallel. An Event Loop is a functionality which handles and processes all your external events and just converts them to a callback function. It invokes all the event handlers at a proper time. Thus, lots of work is done on the back-end, while processing a single request, so that the new incoming request doesn’t have to wait if the processing is not complete.

While processing a request, Node.js attaches a callback function to it and moves it to the back-end. Now, whenever its response is ready, an event is called which triggers the associated callback function to send this response.

**Source:** *techbeamers.com*

**Q16: When should we use Node.js? ☆☆☆**

**Answer:** **Node.js** is well suited for applications that have a lot of concurrent connections and each *request only needs very few CPU cycles*, because the event loop (with all the other clients) is blocked during execution of a function. I believe Node.js is best suited for real-time applications: online games, collaboration tools, chat rooms, or anything where what one user (or robot? or sensor?) does with the application needs to be seen by other users immediately, without a page refresh.

**Source:** *techbeamers.com*

**Q17: How does Node.js handle child threads? ☆☆☆**

**Answer:** Node.js, in its essence, is a single thread process. It does not expose child threads and thread management methods to the developer. Technically, Node.js does spawn child threads for certain tasks such as asynchronous I/O, but these run behind the scenes and do not execute any application JavaScript code, nor block the main event loop.

If threading support is desired in a Node.js application, there are tools available to enable it, such as the ChildProcess module.

**Source:** *lazyquestion.com*

**Q18: What is the preferred method of resolving unhandled exceptions in Node.js? ☆☆☆**

**Answer:** Unhandled exceptions in Node.js can be caught at the Process level by attaching a handler for uncaughtException event.

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

console.log('Caught exception: ' + err);

});

However, uncaughtException is a very crude mechanism for exception handling and may be removed from Node.js in the future. An exception that has bubbled all the way up to the Process level means that your application, and Node.js may be in an undefined state, and the only sensible approach would be to restart everything.

The preferred way is to add another layer between your application and the Node.js process which is called the [domain](http://nodejs.org/api/domain.html).

Domains provide a way to handle multiple different I/O operations as a single group. So, by having your application, or part of it, running in a separate domain, you can safely handle exceptions at the domain level, before they reach the Process level.

**Source:** *lazyquestion.com*

**Q19: What is stream and what are types of streams available in Node.js? ☆☆☆**

**Answer:** **Streams** are a collection of data that might not be available all at once and don’t have to fit in memory. Streams provide chunks of data in a continuous manner. It is useful to read a large set of data and process it.

There is four fundamental type of streams:

* Readable.
* Writeable.
* Duplex.
* Transform.

Readable streams as the name suggest used in reading a large chunk of data from a source. Writable streams are used in writing a large chunk of data to the destination.

Duplex streams are both readable and writable ( Eg socket). Transform stream is the duplex stream which is used in modifying the data (eg zip creation).

**Source:** *codeforgeek.com*

**Q20: What are the global objects of Node.js? ☆☆☆**

**Answer:** These objects are available in all modules:

* **process** - The process object is a global that provides information about, and control over, the current Node.js process.
* **console** - Used to print to stdout and stderr.
* **buffer** - Used to handle binary data.

**Source:** *github.com/jimuyouyou*

**Q1: What is Piping in Node? ☆☆☆☆**

**Answer:** **Piping** is a mechanism to connect output of one stream to another stream. It is normally used to get data from one stream and to pass output of that stream to another stream. There is no limit on piping operations.

**Source:** *tutorialspoint.com*

**Q2: Name some of the events fired by streams. ☆☆☆☆**

**Answer:** Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times. For example, some of the commonly used events are:

* **data** - This event is fired when there is data is available to read.
* **end** - This event is fired when there is no more data to read.
* **error** - This event is fired when there is any error receiving or writing data.
* **finish** - This event is fired when all data has been flushed to underlying system

**Source:** *tutorialspoint.com*

**Q3: What is the purpose of \_\_filename variable? ☆☆☆☆**

**Answer:** 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.

**Source:** *tutorialspoint.com*

**Q4: How can you listen on port 80 with Node? ☆☆☆☆**

**Answer:** Run the application on any port above 1024, then put a reverse proxy like [nginx](http://nginx.org/) in front of it.

**Source:** *blog.risingstack.com*

**Q5: What tools can be used to assure consistent code style? ☆☆☆☆**

**Answer:** You have plenty of options to do so:

* [JSLint](http://jslint.com/) by Douglas Crockford
* [JSHint](http://jshint.com/)
* [ESLint](http://eslint.org/)
* [JSCS](http://jscs.info/)

These tools are really helpful when developing code in teams, to enforce a given style guide and to catch common errors using static analysis.

**Source:** *blog.risingstack.com*

**Q6: What's a stub? Name a use case. ☆☆☆☆**

**Answer:** **Stubs** are functions/programs that simulate the behaviours of components/modules. Stubs provide canned answers to function calls made during test cases. Also, you can assert on with what these stubs were called.

A use-case can be a file read, when you do not want to read an actual file:

var fs = require('fs');

var readFileStub = sinon.stub(fs, 'readFile', function(path, cb) {

return cb(null, 'filecontent');

});

expect(readFileStub).to.be.called;

readFileStub.restore();

**Source:** *blog.risingstack.com*

**Q7: Does Node.js support multi-core platforms? And is it capable of utilizing all the cores? ☆☆☆☆**

**Answer:** Yes, Node.js would run on a multi-core system without any issue. But it is by default a single-threaded application, so it can’t completely utilize the multi-core system.

However, Node.js can facilitate deployment on multi-core systems where it does use the additional hardware. It packages with a Cluster module which is capable of starting multiple Node.js worker processes that will share the same port.

**Source:** *techbeamers.com*

**Q8: Is Node.js entirely based on a single-thread? ☆☆☆☆**

**Answer:** Yes, it’s true that Node.js processes all requests on a single thread. But it’s just a part of the theory behind Node.js design. In fact, more than the single thread mechanism, it makes use of events and callbacks to handle a large no. of requests asynchronously.

Moreover, Node.js has an optimized design which utilizes both JavaScript and C++ to guarantee maximum performance. JavaScript executes at the server-side by Google Chrome v8 engine. And the C++ lib UV library takes care of the non-sequential I/O via background workers.

To explain it practically, let’s assume there are 100s of requests lined up in Node.js queue. As per design, the main thread of Node.js event loop will receive all of them and forwards to background workers for execution. Once the workers finish processing requests, the registered callbacks get notified on event loop thread to pass the result back to the user.

**Source:** *techbeamers.com*

**Q9: Is Node.js entirely based on a single-thread? ☆☆☆☆**

**Answer:** Yes, it’s true that Node.js processes all requests on a single thread. But it’s just a part of the theory behind Node.js design. In fact, more than the single thread mechanism, it makes use of events and callbacks to handle a large no. of requests asynchronously.

Moreover, Node.js has an optimized design which utilizes both JavaScript and C++ to guarantee maximum performance. JavaScript executes at the server-side by Google Chrome v8 engine. And the C++ lib UV library takes care of the non-sequential I/O via background workers.

To explain it practically, let’s assume there are 100s of requests lined up in Node.js queue. As per design, the main thread of Node.js event loop will receive all of them and forwards to background workers for execution. Once the workers finish processing requests, the registered callbacks get notified on event loop thread to pass the result back to the user.

**Source:** *techbeamers.com*

**Q10: When to not use Node.js? ☆☆☆☆**

**Answer:** We can use Node.js for a variety of applications. But it is a single threaded framework, so we should not use it for cases where the application requires long processing time. If the server is doing some calculation, it won’t be able to process any other requests. Hence, Node.js is best when processing needs less dedicated CPU time.

**Source:** *techbeamers.com*

**Q11: Why to use Buffers instead of binary strings to handle binary data ? ☆☆☆☆**

**Answer:** Pure JavaScript does not able to handle straight binary data very well. Since Node.js servers have to deal with TCP streams for reading and writing of data, binary strings will become problematic to work with as it is very slow and has a tendency to break. That's why it is always advisable to use Buffers instead of binary strings to handle binary data.

**Source:** *codingdefined.com*

**Q12: How to use Buffer in Node.js? ☆☆☆**

**Answer:** Buffer is used to process binary data, such as pictures, mp3, database files, etc. Buffer supports a variety of encoding and decoding, binary string conversion.

**Source:** *github.com/jimuyouyou*

**Q13: When should I use EventEmitter? ☆☆☆**

**Answer:** Whenever it makes sense for code to *subscribe* to something rather than get a callback from something. The typical use case would be that there's multiple blocks of code in your application that may need to do something when an event happens.

**Source:** *stackoverflow.com/*

**Q14: How do you debug Node.js applications? ☆☆☆**

**Answer:** Node has its own built in GUI debugger as of version 6.3 (using Chrome's DevTools).

node --inspect server.js

Some other options for debugging are:

* [Joyent's Guide](http://www.joyent.com/developers/node/debug)
* [Debugger](https://nodejs.org/api/debugger.html)
* [Node Inspector](https://github.com/node-inspector/node-inspector)
* [Visual Studio Code](https://code.visualstudio.com/docs/runtimes/nodejs#_debugging-your-node-application)
* [Cloud9](https://docs.c9.io/running_and_debugging_code.html)
* [Brackets](https://github.com/adobe-research/theseus)

**Source:** *stackoverflow.com*

**Q15: Rewrite promise-based Node.js applications to Async/Await ☆☆☆**

**Details:** Rewrite this code to Async/Await:

function asyncTask() {

return functionA()

.then((valueA) => functionB(valueA))

.then((valueB) => functionC(valueB))

.then((valueC) => functionD(valueC))

.catch((err) => logger.error(err))

}

**Answer:**

async function asyncTask() {

try {

const valueA = await functionA()

const valueB = await functionB(valueA)

const valueC = await functionC(valueB)

return await functionD(valueC)

} catch (err) {

logger.error(err)

}

}

**Source:** *stackoverflow.com*

**Q16: What is the relationship between Node.js and V8? ☆☆☆**

**Answer:** V8 is the Javascript engine inside of node.js that parses and runs your Javascript. The same V8 engine is used inside of Chrome to run javascript in the Chrome browser. Google open-sourced the V8 engine and the builders of node.js used it to run Javascript in node.js.

**Source:** *stackoverflow.com*

**Q17: What is N-API in Node.js? ☆☆☆**

**Answer:** **N-API** (pronounced N as in the letter, followed by API) is an API for building native Addons. It is independent from the underlying JavaScript runtime (ex V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate Addons from changes in the underlying JavaScript engine and allow modules compiled for one version to run on later versions of Node.js without recompilation.

**Source:** *medium.com*

**Q18: Explain the concept of Domain in Node.js ☆☆☆**

**Answer:** Domains provide a way to handle multiple different IO operations as a single group. If any of the event emitters or callbacks registered to a domain emit an error event, or throw an error, then the domain object will be notified, rather than losing the context of the error in the process.on('uncaughtException') handler, or causing the program to exit immediately with an error code.

Domain error handlers are not a substitute for closing down a process when an error occurs. The safest way to respond to a thrown error is to shut down the process. In a normal web server, the better approach is to send an error response to the request that triggered the error, while letting the others finish in their normal time, and stop listening for new requests in that worker.

var domain = require('domain');

var d = domain.create();

// Domain emits 'error' when it's given an unhandled error

d.on('error', function(err) {

console.log(err.stack);

// Our handler should deal with the error in an appropriate way

});

// Enter this domain

d.run(function() {

// If an un-handled error originates from here, process.domain will handle it

console.log(process.domain === d); // true

});

// domain has now exited. Any errors in code past this point will not be caught.

**Source:** *nodejs.org*

**Q19: Are you familiar with differences between Node.js nodules and ES6 nodules? ☆☆☆**

**Answer:** The modules used in Node.js follow a module specification known as the **CommonJS** specification. The recent updates to the JavaScript programming language, in the form of ES6, specify changes to the language, adding things like new class syntax and a module system. This module system is different from Node.js modules. To import ES6 module, we'd use the ES6 import functionality.

Now ES6 modules are incompatible with Node.js modules. This has to do with the way modules are loaded differently between the two formats. If you use a compiler like Babel, you can mix and match module formats.

**Source:** *stackoverflow.com*

**Q20: What are the use cases for the Node.js "vm" core module? ☆☆☆**

**Answer:** It can be used to safely execute a piece of code contained in a string or file. The execution is performed in a separate environment that by default has no access to the environment of the program that created it. Moreover, you can specify execution timeout and context-specific error handling.

**Source:** *quora.com*

**Q1: What is Piping in Node? ☆☆☆☆**

**Answer:** **Piping** is a mechanism to connect output of one stream to another stream. It is normally used to get data from one stream and to pass output of that stream to another stream. There is no limit on piping operations.

**Source:** *tutorialspoint.com*

**Q2: Name some of the events fired by streams. ☆☆☆☆**

**Answer:** Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times. For example, some of the commonly used events are:

* **data** - This event is fired when there is data is available to read.
* **end** - This event is fired when there is no more data to read.
* **error** - This event is fired when there is any error receiving or writing data.
* **finish** - This event is fired when all data has been flushed to underlying system

**Source:** *tutorialspoint.com*

**Q3: What is the purpose of \_\_filename variable? ☆☆☆☆**

**Answer:** 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.

**Source:** *tutorialspoint.com*

**Q4: How can you listen on port 80 with Node? ☆☆☆☆**

**Answer:** Run the application on any port above 1024, then put a reverse proxy like [nginx](http://nginx.org/) in front of it.

**Source:** *blog.risingstack.com*

**Q5: What tools can be used to assure consistent code style? ☆☆☆☆**

**Answer:** You have plenty of options to do so:

* [JSLint](http://jslint.com/) by Douglas Crockford
* [JSHint](http://jshint.com/)
* [ESLint](http://eslint.org/)
* [JSCS](http://jscs.info/)

These tools are really helpful when developing code in teams, to enforce a given style guide and to catch common errors using static analysis.

**Source:** *blog.risingstack.com*

**Q6: What's a stub? Name a use case. ☆☆☆☆**

**Answer:** **Stubs** are functions/programs that simulate the behaviours of components/modules. Stubs provide canned answers to function calls made during test cases. Also, you can assert on with what these stubs were called.

A use-case can be a file read, when you do not want to read an actual file:

var fs = require('fs');

var readFileStub = sinon.stub(fs, 'readFile', function(path, cb) {

return cb(null, 'filecontent');

});

expect(readFileStub).to.be.called;

readFileStub.restore();

**Source:** *blog.risingstack.com*

**Q7: Does Node.js support multi-core platforms? And is it capable of utilizing all the cores? ☆☆☆☆**

**Answer:** Yes, Node.js would run on a multi-core system without any issue. But it is by default a single-threaded application, so it can’t completely utilize the multi-core system.

However, Node.js can facilitate deployment on multi-core systems where it does use the additional hardware. It packages with a Cluster module which is capable of starting multiple Node.js worker processes that will share the same port.

**Source:** *techbeamers.com*

**Q8: Is Node.js entirely based on a single-thread? ☆☆☆☆**

**Answer:** Yes, it’s true that Node.js processes all requests on a single thread. But it’s just a part of the theory behind Node.js design. In fact, more than the single thread mechanism, it makes use of events and callbacks to handle a large no. of requests asynchronously.

Moreover, Node.js has an optimized design which utilizes both JavaScript and C++ to guarantee maximum performance. JavaScript executes at the server-side by Google Chrome v8 engine. And the C++ lib UV library takes care of the non-sequential I/O via background workers.

To explain it practically, let’s assume there are 100s of requests lined up in Node.js queue. As per design, the main thread of Node.js event loop will receive all of them and forwards to background workers for execution. Once the workers finish processing requests, the registered callbacks get notified on event loop thread to pass the result back to the user.

**Source:** *techbeamers.com*

**Q9: Is Node.js entirely based on a single-thread? ☆☆☆☆**

**Answer:** Yes, it’s true that Node.js processes all requests on a single thread. But it’s just a part of the theory behind Node.js design. In fact, more than the single thread mechanism, it makes use of events and callbacks to handle a large no. of requests asynchronously.

Moreover, Node.js has an optimized design which utilizes both JavaScript and C++ to guarantee maximum performance. JavaScript executes at the server-side by Google Chrome v8 engine. And the C++ lib UV library takes care of the non-sequential I/O via background workers.

To explain it practically, let’s assume there are 100s of requests lined up in Node.js queue. As per design, the main thread of Node.js event loop will receive all of them and forwards to background workers for execution. Once the workers finish processing requests, the registered callbacks get notified on event loop thread to pass the result back to the user.

**Source:** *techbeamers.com*

**Q10: When to not use Node.js? ☆☆☆☆**

**Answer:** We can use Node.js for a variety of applications. But it is a single threaded framework, so we should not use it for cases where the application requires long processing time. If the server is doing some calculation, it won’t be able to process any other requests. Hence, Node.js is best when processing needs less dedicated CPU time.

**Source:** *techbeamers.com*

**Q11: Why to use Buffers instead of binary strings to handle binary data ? ☆☆☆☆**

**Answer:** Pure JavaScript does not able to handle straight binary data very well. Since Node.js servers have to deal with TCP streams for reading and writing of data, binary strings will become problematic to work with as it is very slow and has a tendency to break. That's why it is always advisable to use Buffers instead of binary strings to handle binary data.

**Source:** *codingdefined.com*

**Q12: How to gracefully Shutdown Node.js Server? ☆☆☆☆**

**Answer:** We can gracefully shutdown Node.js server by using the generic signal called SIGTERM or SIGINT which is used for program termination. We need to call SIGTERM or SIGINT which will terminate the program and clean up the resources utilized by the program.

**Source:** *codingdefined.com*

**Q13: What are the timing features of Node.js? ☆☆☆☆**

**Answer:** The Timers module in Node.js contains functions that execute code after a set period of time.

* **setTimeout/clearTimeout** - can be used to schedule code execution after a designated amount of milliseconds
* **setInterval/clearInterval** - can be used to execute a block of code multiple times
* **setImmediate/clearImmediate** - will execute code at the end of the current event loop cycle
* **process.nextTick** - used to schedule a callback function to be invoked in the next iteration of the Event Loop

function cb(){

console.log('Processed in next iteration');

}

process.nextTick(cb);

console.log('Processed in the first iteration');

Output:

Processed in the first iteration

Processed in next iteration

**Source:** *github.com/jimuyouyou*

**Q14: Explain usage of NODE\_ENV ☆☆☆☆**

**Answer:** Node encourages the convention of using a variable called NODE\_ENV to flag whether we’re in production right now. This determination allows components to provide better diagnostics during development, for example by disabling caching or emitting verbose log statements. Setting NODE\_ENV to production makes your application 3 times faster.

// Setting environment variables in bash before starting the node process

$ NODE\_ENV=development

$ node

// Reading the environment variable using code

if (process.env.NODE\_ENV === “production”)

useCaching = true;

**Source:** *github.com/i0natan/nodebestpractices*

**Q15: What is LTS releases of Node.js why should you care? ☆☆☆☆**

**Answer:** An **LTS(Long Term Support)** version of Node.js receives all the critical bug fixes, security updates and performance improvements.

LTS versions of Node.js are supported for at least 18 months and are indicated by even version numbers (e.g. 4, 6, 8). They're best for production since the LTS release line is focussed on stability and security, whereas the *Current* release line has a shorter lifespan and more frequent updates to the code. Changes to LTS versions are limited to bug fixes for stability, security updates, possible npm updates, documentation updates and certain performance improvements that can be demonstrated to not break existing applications.

**Source:** *github.com/i0natan/nodebestpractices*

**Q16: Provide some example of config file separation for dev and prod environments ☆☆☆☆**

**Answer:** A perfect and flawless configuration setup should ensure:

* keys can be read from file AND from environment variable
* secrets are kept outside committed code
* config is hierarchical for easier findability

Consider the following config file:

var config = {

production: {

mongo : {

billing: '\*\*\*\*'

}

},

default: {

mongo : {

billing: '\*\*\*\*'

}

}

}

exports.get = function get(env) {

return config[env] || config.default;

}

And it's usage:

const config = require('./config/config.js').get(process.env.NODE\_ENV);

const dbconn = mongoose.createConnection(config.mongo.billing);

**Source:** *github.com/i0natan/nodebestpractices*

**Q17: How would you handle errors for async code in Node.js? ☆☆☆☆**

**Answer:** Handling async errors in callback style (error-first approach) is probably the fastest way to hell (a.k.a the pyramid of doom). It's better to use a reputable promise library or async-await instead which enables a much more compact and familiar code syntax like try-catch.

Consider promises to catch errors:

doWork()

.then(doWork)

.then(doOtherWork)

.then((result) => doWork)

.catch((error) => {throw error;})

.then(verify);

or using async/await:

async function check(req, res) {

try {

const a = await someOtherFunction();

const b = await somethingElseFunction();

res.send("result")

} catch (error) {

res.send(error.stack);

}

}

**Source:** *github.com/i0natan/nodebestpractices*

**Q18: What's the difference between dependencies, devDependencies and peerDependencies in npm package.json file? ☆☆☆☆**

**Answer:**

* **dependencies** - Dependencies that your project needs to run, like a library that provides functions that you call from your code. They are installed transitively (if A depends on B depends on C, npm install on A will install B and C).
* **devDependencies** - Dependencies you only need during development or releasing, like compilers that take your code and compile it into javascript, test frameworks or documentation generators. They are not installed transitively (if A depends on B dev-depends on C, npm install on A will install B only).
* **peerDependencies** - Dependencies that your project hooks into, or modifies, in the parent project, usually a plugin for some other library or tool. It is just intended to be a check, making sure that the parent project (project that will depend on your project) has a dependency on the project you hook into. So if you make a plugin C that adds functionality to library B, then someone making a project A will need to have a dependency on B if they have a dependency on C. They are not installed (unless npm < 3), they are only checked for.

**Source:** *stackoverflow.com*

**Q19: How do you convert an existing callback API to promises? ☆☆☆☆**

**Details:** How to convert this callback code to Promise? Provide some examples.

function divisionAPI (number, divider, successCallback, errorCallback) {

if (divider == 0) {

return errorCallback( new Error("Division by zero") )

}

successCallback( number / divider )

}

**Answer:**

function divisionAPI(number, divider) {

return new Promise(function(fulfilled, rejected) {

if (divider == 0) {

return rejected(new Error("Division by zero"))

}

fulfilled(number / divider)

})

}

// Promise can be used with together async\await in ES7 to make the program flow wait for a fullfiled result

async function foo() {

var result = await divisionAPI(1, 2); // awaits for a fulfilled result!

console.log(result);

}

// Another usage with the same code by using .then() method

divisionAPI(1, 2).then(function(result) {

console.log(result)

})

Node.js 8.0.0 includes a new util.promisify() API that allows standard Node.js callback style APIs to be wrapped in a function that returns a Promise.

const fs = require('fs');

const util = require('util');

const readfile = util.promisify(fs.readFile);

readfile('/some/file')

.then((data) => {

/\*\* ... \*\*/

})

.catch((err) => {

/\*\* ... \*\*/

});

**Source:** *stackoverflow.com*

**Q20: What are async functions in Node? Provide some examples. ☆☆☆☆**

**Answer:** With the release of Node.js 8, the long awaited async functions have landed in Node.js as well. ES 2017 introduced Asynchronous functions. Async functions are essentially a cleaner way to work with asynchronous code in JavaScript.

Async/Await is:

* The newest way to write asynchronous code in JavaScript.
* It is non blocking (just like promises and callbacks).
* Async/Await was created to simplify the process of working with and writing chained promises.
* Async functions return a Promise. If the function throws an error, the Promise will be rejected. If the function returns a value, the Promise will be resolved.

**Async functions** can make use of the await expression. This will pause the async function and wait for the *Promise* to resolve prior to moving on.

**Q1: Consider following code snippet ☆☆☆☆☆**

**Details:** Consider following code snippet:

{

console.time("loop");

for (var i = 0; i < 1000000; i += 1) {

// Do nothing

}

console.timeEnd("loop");

}

The time required to run this code in Google Chrome is considerably more than the time required to run it in Node.js Explain why this is so, even though both use the v8 JavaScript Engine.

**Answer:** Within a web browser such as Chrome, declaring the variable i outside of any function’s scope makes it global and therefore binds it as a property of the window object. As a result, running this code in a web browser requires repeatedly resolving the property i 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’s own scope (not the window object) which therefore makes it much easier and faster to resolve.

**Source:** *toptal.com*

**Q2: Can Node.js use other engines than V8? ☆☆☆☆☆**

**Answer:** Yes. Microsoft Chakra is another JavaScript engine which can be used with Node.js. It’s not officially declared yet.

**Source:** *codeforgeek.com*

**Q3: How would you scale Node application? ☆☆☆☆☆**

**Answer:** We can scale Node application in following ways:

* cloning using *Cluster* module.
* decomposing the application into smaller services – i.e micro services.

**Source:** *codeforgeek.com*

**Q4: What is the difference between process.nextTick() and setImmediate() ? ☆☆☆☆☆**

**Answer:** The difference between process.nextTick() and setImmediate() is that process.nextTick() defers the execution of an action till the next pass around the event loop or it simply calls the callback function once the ongoing execution of the event loop is finished whereas setImmediate() executes a callback on the next cycle of the event loop and it gives back to the event loop for executing any I/O operations.

**Source:** *codingdefined.com*

**Q5: How to solve "Process out of Memory Exception" in Node.js ? ☆☆☆☆☆**

**Answer:** To solve the process out of memory exception in Node.js we need to increase the max-old-space-size. By default the max size of max-old-space-size is 512 mb which you can increase by the command:

node --max-old-space-size=1024 file.js

**Source:** *codingdefined.com*

**Q6: Explain what is Reactor Pattern in Node.js? ☆☆☆☆☆**

**Answer:** **Reactor Pattern** is an idea of non-blocking I/O operations in Node.js. This pattern provides a handler(in case of Node.js, a *callback function*) that is associated with each I/O operation. When an I/O request is generated, it is submitted to a *demultiplexer*.

This *demultiplexer* is a notification interface that is used to handle concurrency in non-blocking I/O mode and collects every request in form of an event and queues each event in a queue. Thus, the demultiplexer provides the *Event Queue*.

At the same time, there is an Event Loop which iterates over the items in the Event Queue. Every event has a callback function associated with it, and that callback function is invoked when the Event Loop iterates.

**Source:** *hackernoon.com*

**Q7: Explain some Error Handling approaches in Node.js you know about. Which one will you use? ☆☆☆☆☆**

**Answer:** Error handling in an asynchronous language works in a unique way and presents many challenges, some unexpected. There are four main error handling patterns in node:

* **Error return value** - doesn't work asynchronously

var validateObject = function (obj) {

if (typeof obj !== 'object') {

return new Error('Invalid object');

}

};

* **Error throwing** - well-establish pattern, in which a function does its thing and if an error situation arises, it simply bails out throwing an error. Can leave you in an unstable state. It requires extra work to catch them. Also wrapping the async calls in try/catch won't help because the errors happen asynchronously. To fix this, we need *domains*. Domains provide an asynchronous try...catch.

var validateObject = function (obj) {

if (typeof obj !== 'object') {

throw new Error('Invalid object');

}

};

try {

validateObject('123');

}

catch (err) {

console.log('Thrown: ' + err.message);

}

* **Error callback** - returning an error via a callback is the most common error handling pattern in Node.js. Handling error callbacks can become a mess (callback hell or the pyramid of doom).

var validateObject = function (obj, callback) {

if (typeof obj !== 'object') {

return callback(new Error('Invalid object'));

}

return callback();

};

validateObject('123', function (err) {

console.log('Callback: ' + err.message);

});

* **Error emitting** - when emitting errors, the errors are broadcast to any interested subscribers and handled within the same process tick, in the order subscribed.

var Events = require('events');

var emitter = new Events.EventEmitter();

var validateObject = function (a) {

if (typeof a !== 'object') {

emitter.emit('error', new Error('Invalid object'));

}

};

emitter.on('error', function (err) {

console.log('Emitted: ' + err.message);

});

validateObject('123');

* **Promises** for async error handling

doWork()

.then(doWork)

.then(doError)

.then(doWork)

.catch(errorHandler)

.then(verify);

* **Try...catch with async/await** - ES7 Async/await allows us as developers to write asynchronous JS code that look synchronous.

async function f() {

try {

let response = await fetch('http://no-such-url');

} catch(err) {

alert(err); // TypeError: failed to fetch

}

}

f();

* **Await-to-js lib** - async/await without try-catch blocks in Javascript

import to from 'await-to-js';

async function main(callback) {

const [err,quote] = await to(getQuote());

if(err || !quote) return callback(new Error('No Quote found');

callback(null,quote);

}

**Source:** *gist.github.com*

**Q8: Why should you separate Express 'app' and 'server'? ☆☆☆☆☆**

**Answer:** Keeping the API declaration separated from the network related configuration (port, protocol, etc) allows testing the API in-process, without performing network calls, with all the benefits that it brings to the table: fast testing execution and getting coverage metrics of the code. It also allows deploying the same API under flexible and different network conditions. Bonus: better separation of concerns and cleaner code.

API declaration, should reside in app.js:

var app = express();

app.use(bodyParser.json());

app.use("/api/events", events.API);

app.use("/api/forms", forms);

Server network declaration, should reside in /bin/www:

var app = require('../app');

var http = require('http');

/\*\*

\* Get port from environment and store in Express.

\*/

var port = normalizePort(process.env.PORT || '3000');

app.set('port', port);

/\*\*

\* Create HTTP server.

\*/

var server = http.createServer(app);

**Source:** *github.com/i0natan/nodebestpractices*

**Q9: Rewrite the code sample without try/catch block ☆☆☆☆☆**

**Details:** Consider the code:

async function check(req, res) {

try {

const a = await someOtherFunction();

const b = await somethingElseFunction();

res.send("result")

} catch (error) {

res.send(error.stack);

}

}

Rewrite the code sample without try/catch block.

**Answer:**

async function getData(){

const a = await someFunction().catch((error)=>console.log(error));

const b = await someOtherFunction().catch((error)=>console.log(error));

if (a && b) console.log("some result")

}

or if you wish to know which specific function caused error:

async function loginController() {

try {

const a = await loginService().

catch((error) => {

throw new CustomErrorHandler({

code: 101,

message: "a failed",

error: error

})

});

const b = await someUtil().

catch((error) => {

throw new CustomErrorHandler({

code: 102,

message: "b failed",

error: error

})

});

//someoeeoe

if (a && b) console.log("no one failed")

} catch (error) {

if (!(error instanceof CustomErrorHandler)) {

console.log("gen error", error)

}

}

}

**Source:** *medium.com*

**Q10: How many threads does Node actually create? ☆☆☆☆☆**

**Answer:** **4 extra threads** are for use by V8. V8 uses these threads to perform various tasks, such as GC-related background tasks and optimizing compiler tasks.

**Source:** *stackoverflow.com*

**Q11: Can Node.js work without V8? ☆☆☆☆**

**Answer:** No. The current node.js binary cannot work without V8. It would have no Javascript engine and thus no ability to run code which would obviously render it non-functional. Node.js was not designed to run with any other Javascript engine and, in fact, all the native code bindings that come with node.js (such as the fs module or the net module) all rely on the specific V8 interface between C++ and Javascript.

There is an effort by Microsoft to allow the Chakra Javascript engine (that's the engine in Edge) to be used with node.js. Node.js can actually function to some extent without V8, through use of the node-chakracore project. There is ongoing work to reduce the tight coupling between V8 and Node, so that different JavaScript engines can be used in-place.

**Source:** *stackoverflow.com*

**Q12: How the V8 engine works? ☆☆☆☆**

**Answer:** **V8** is a JavaScript engine built at the google development center, in Germany. It is open source and written in C++. It is used for both client side (Google Chrome) and server side (node.js) JavaScript applications.

V8 was first designed to increase the performance of the JavaScript execution inside web browsers. In order to obtain speed, V8 translates JavaScript code into more efficient machine code instead of using an interpreter. It compiles JavaScript code into machine code at execution by implementing a **JIT (Just-In-Time)** compiler like a lot of modern JavaScript engines such as SpiderMonkey or Rhino (Mozilla) are doing. The main difference with V8 is that it doesn’t produce bytecode or any intermediate code.

**Source:** *nodejs.org*

**Q13: What is the purpose of using hidden classes in V8? ☆☆☆☆☆**

**Answer:** JavaScript is a prototype-based language: there are no classes and objects are created by using a cloning process. JavaScript is also dynamically typed: types and type informations are not explicit and properties can be added to and deleted from objects on the fly.

Accessing types and properties effectively makes a first big challenge for V8. Instead of using a dictionary-like data structure for storing object properties and doing a dynamic lookup to resolve the property location (like most JavaScript engines do), V8 creates\*\* hidden classes\*\*, at runtime, in order to have an internal representation of the type system and to improve the property access time.

**Source:** *thibaultlaurens.github.io*

**Q14: How V8 compiles JavaScript code? ☆☆☆☆☆**

**Answer:** V8 has two compilers:

* A **“Full” Compiler** that can generate good code for any JavaScript: good but not great JIT code. The goal of this compiler is to generate code quickly. To achieve its goal, it doesn’t do any type analysis and doesn’t know anything about types. Instead, it uses an Inline Caches or “IC” strategy to refine knowledge about types while the program runs. IC is very efficient and brings about 20 times speed improvment.
* An **Optimizing Compiler** that produces great code for most of the JavaScript language. It comes later and re-compiles hot functions. The optimizing compiler takes types from the Inline Cache and make decisions about how to optimize the code better. However, some language features are not supported yet like try/catch blocks for instance. (The workaround for try/catch blocks is to write the “non stable” code in a function and call the function in the try block)

V8 also supports **de-optimization**: the optimizing compiler makes optimistic assumptions from the Inline Cache about the different types, de-optimization comes if these assumptions are invalid. For example, if a hidden class generated was not the one expected, V8 throws away the optimized code and comes back to the Full Compiler to get types again from the Inline Cache. This process is slow and should be avoided by trying to not change functions after they are optimized.

**Source:** *thibaultlaurens.github.io*

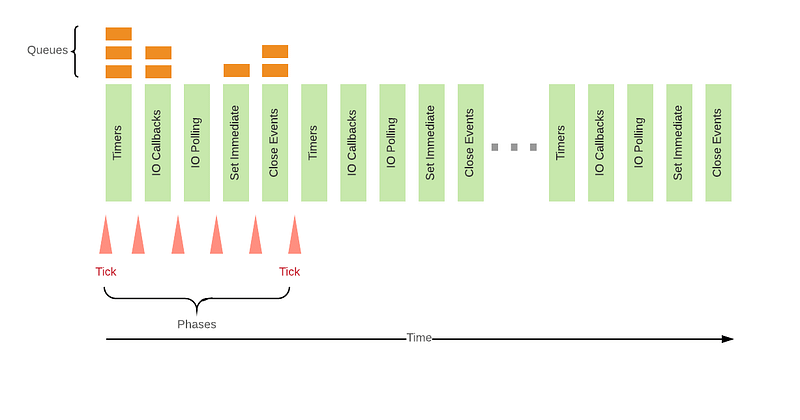
**Q15: How does libuv work under the hood? ☆☆☆☆☆**

**Answer:** There is only one thread that executes JavaScript code and this is the thread where the event loop is running provided by **libuv**. The execution of callbacks (know that every userland code in a running Node.js application is a callback) is done by the event loop.

Libuv by default creates a thread pool with four threads to offload asynchronous work to. Today’s operating systems already provide asynchronous interfaces for many I/O tasks (e.g. AIO on Linux). Whenever possible, libuv will use those asynchronous interfaces, avoiding usage of the thread pool.

The event loop as a process is a set of phases with specific tasks that are processed in a round-robin manner. Each phase has a FIFO queue of callbacks to execute. While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

* **timers**: this phase executes callbacks scheduled by setTimeout() and setInterval().
* **pending callbacks**: executes I/O callbacks deferred to the next loop iteration.
* **idle, prepare**: only used internally.
* **poll**: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and setImmediate()); node will block here when appropriate.
* **check**: setImmediate() callbacks are invoked here.
* **close callbacks**: some close callbacks, e.g. socket.on('close', ...).

[](https://camo.githubusercontent.com/647a2827056e70da3dd836d23a2f0631bea90820e3fd3a0a00fa20312a36142b/68747470733a2f2f63646e2d696d616765732d312e6d656469756d2e636f6d2f6d61782f3830302f312a524f786961767a374c655270496663675244453743412e706e67)

**Source:** *nodejs.org*

**Q16: How does the cluster module work? What’s the difference between it and a load balancer? ☆☆☆☆**

**Answer:** The cluster module performs fork from your server (at that moment it is already an OS process), thus creating several slave processes. The cluster module supports two methods of distributing incoming connections.

* The first one (and the default one on all platforms except Windows), is the round-robin approach, where the master process listens on a port, accepts new connections and distributes them across the workers in a round-robin fashion, with some built-in smarts to avoid overloading a worker process.
* The second approach is where the master process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly.

The difference between a cluster module and a load balancer is that instead of distributing load between processes, the balancer distributes requests.

**Source:** *imasters.com*

**Q17: What is V8 Templates? ☆☆☆☆☆**

**Answer:** A template is a blueprint for JavaScript functions and objects. You can use a template to wrap C++ functions and data structures within JavaScript objects. V8 has two types of templates: Function Templates and Object Templates.

* **Function Template** is the blueprint for a single function. You create a JavaScript instance of template by calling the template’s GetFunction method from within the context in which you wish to instantiate the JavaScript function. You can also associate a C++ callback with a function template which is called when the JavaScript function instance is invoked.
* **Object Template** is used to configure objects created with function template as their constructor. You can associate two types of C++ callbacks with object templates: accessor callback and interceptor callback. Accessor callback is invoked when a specific object property is accessed by a script. Interceptor callback is invoked when any object property is accessed by a script. In a nutshell, you can wrap C++ objects\structures within JavaScript objects.

**Source:** *blog.ghaiklor.com*

**Q18: Why do we need C++ Addons in Node.js? ☆☆☆☆☆**

**Answer:** **Node.js Addons** are dynamically-linked shared objects, written in C++, that can be loaded into Node.js using the require() function, and used just as if they were an ordinary Node.js module. They are used primarily to provide an interface between JavaScript running in Node.js and C/C++ libraries.

There can be many reasons to write nodejs addons:

1. You may want to access some native apis that is difficult using JS alone.
2. You may want to integrate a third party library written in C/C++ and use it directly in Node.js.
3. You may want to rewrite some of the modules in C++ for performance reasons.

N-API (pronounced N as in the letter, followed by API) is an API for building native Addons.

**Source:** *nodejs.org*

**Q19: Is it possible to use "Class" in Node.js? ☆☆☆☆**

**Answer:** With ES6, you are able to make "actual" classes just like this:

class Animal {

constructor(name) {

this.name = name;

}

print() {

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

}

}

You can export a class just like anything else:

module.exports = class Animal {

};

Once imported into another module, then you can treat it as if it were defined in that file:

var Animal = require('./Animal');

class Cat extends Animal {

...

}

**Source:** *stackoverflow.com*

**Q20: Why Node.js devs tend to lean towards the Module Requiring vs Dependency Injection? ☆☆☆☆☆**

**Answer:** Dependency injection is somewhat the opposite of normal *module design*. In normal module design, a module uses require() to load in all the other modules that it needs with the goal of making it simple for the caller to use your module. The caller can just require() in your module and your module will load all the other things it needs.

With dependency injection, rather than the module loading the things it needs, the caller is required to pass in things (usually objects) that the module needs. This can make certain types of testing easier and it can make mocking certain things for testing purposes easier.

Modules and dependency injection are orthogonal: if you need dependency injection for testability or extensibility then use it. If not, importing modules is fine. The great thing about JS is that you can modify just about anything to achieve what you want. This comes in handy when it comes to testing.

**Source:** *reddit.com*

**Q1: Explain the result of this code execution ☆☆☆☆☆**

**Details:** Explain the result of that code execution:

var EventEmitter = require("events");

var crazy = new EventEmitter();

crazy.on('event1', function () {

console.log('event1 fired!');

crazy.emit('event2');

});

crazy.on('event2', function () {

console.log('event2 fired!');

crazy.emit('event3');

});

crazy.on('event3', function () {

console.log('event3 fired!');

crazy.emit('event1');

});

crazy.emit('event1');

**Answer:** You’ll get an exception that basically says the call stack has exploded. Why? Every emit will invoke synchronous code. Because all callbacks are executed in a synchronous manner it’ll just recursive call itself to infinity and beyond.

Output:

console.js:165

if (isStackOverflowError(e))

^

RangeError: Maximum call stack size exceeded

at write (console.js:165:9)

at Console.log (console.js:197:3)

at EventEmitter.<anonymous> (C:\Work\Node\main.js:6:13)

at EventEmitter.emit (events.js:182:13)

at EventEmitter.<anonymous> (C:\Work\Node\main.js:18:11)

at EventEmitter.emit (events.js:182:13)

at EventEmitter.<anonymous> (C:\Work\Node\main.js:12:11)

at EventEmitter.emit (events.js:182:13)

at EventEmitter.<anonymous> (C:\Work\Node\main.js:7:11)

at EventEmitter.emit (events.js:182:13)

**Source:** *codementor.io*

**Q2: Explain the result of this code execution ☆☆☆☆☆**

**Details:** Explain the result of this code execution

var EventEmitter = require('events');

var crazy = new EventEmitter();

crazy.on('event1', function () {

console.log('event1 fired!');

setImmediate(function () {

crazy.emit('event2');

});

});

crazy.on('event2', function () {

console.log('event2 fired!');

setImmediate(function () {

crazy.emit('event3');

});

});

crazy.on('event3', function () {

console.log('event3 fired!');

setImmediate(function () {

crazy.emit('event1');

});

});

crazy.emit('event1');

**Answer:** Shortly - the app will be run infinitely. Any function passed as the setImmediate() argument is a callback that's executed in the *next iteration* of the event loop. Without setImmidiate all callbacks are executed in a synchronous manner. With setImmidiate each call back executed as a part of next event loop iteration so no recursion/stuck occurs.

**Source:** *codementor.io*

**Q3: What will happen when that code will be executed? ☆☆☆☆☆**

**Details:**

What will happen when that code will be executed?

var EventEmitter = require('events');

var crazy = new EventEmitter();

crazy.on('event1', function () {

console.log('event1 fired!');

process.nextTick(function () {

crazy.emit('event2');

});

});

crazy.on('event2', function () {

console.log('event2 fired!');

process.nextTick(function () {

crazy.emit('event3');

});

});

crazy.on('event3', function () {

console.log('event3 fired!');

process.nextTick(function () {

crazy.emit('event1');

});

});

crazy.emit('event1');

**Answer:** It’ll get stuck! And if you wait long enough, about 30 seconds, it’ll eventually give you a “process out of memory” exception. Now, the problem is not stack overflow, it’s GC not being able to reclaim memory. Every handler has its own closure to access the crazy on the outer layer. This cost comes out of the heap. Though you might not be 100% why GC can't successfully get the memory back, you can probably guess that the program got stuck in some even loop phase because there’s always another process.nextTick callback to be processed. So essentially, the event loop is blocked completely.

**Source:** \_codementor.io\_Consider the code:

function doubleAfter2Seconds(x) {

return new Promise(resolve => {

setTimeout(() => {

resolve(x \* 2);

}, 2000);

});

}

What if we want to run a few different values through our function and add the result?

Promise-based solution will be:

function addPromise(x) {

return new Promise(resolve => {

doubleAfter2Seconds(10).then((a) => {

doubleAfter2Seconds(20).then((b) => {

doubleAfter2Seconds(30).then((c) => {

resolve(x + a + b + c);

})

})

})

});

}

addPromise(10).then((sum) => {

console.log(sum);

});

Async/Await solution will look like:

async function addAsync(x) {

const a = await doubleAfter2Seconds(10);

const b = await doubleAfter2Seconds(20);

const c = await doubleAfter2Seconds(30);

return x + a + b + c;

}

addAsync(10).then((sum) => {

console.log(sum);

});