Node js

node js interview questions
1 usage of socket
2 aws services asked
3 usage of lambda functions done
4 how do you define node js
5 create own code for 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. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.
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.

```
const express = require('express')
const app = express()
app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.listen(3000)
```

Cluster to increase Node.js performance
The cluster module provides a way of creating child processes that runs simultaneously and share the same server port.
Node.js runs single threaded programming, which is very memory efficient, but to take advantage of computers multi-core systems, the Cluster module allows you to easily create child processes that each runs on their own single thread, to handle the load.
The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
Benefits of Promises
Improves Code Readability
Better handling of asynchronous operations
Better flow of control definition in asynchronous logic
Better Error Handling
A Promise has four states:
fulfilled: Action related to the promise succeeded
rejected: Action related to the promise failed
pending: Promise is still pending i.e. not fulfilled or rejected yet
settled: Promise has fulfilled or rejected
A promise can be created using Promise constructor.

Parameters
Promise constructor takes only one argument which is a callback function (and that callback function is also referred as anonymous function too).
Callback function takes two arguments, resolve and reject
Perform operations inside the callback function and if everything went well then call resolve.
If desired operations do not go well then call reject.
Promise Consumers
Promises can be consumed by registering functions using .then and .catch methods.
1. then()
then() is invoked when a promise is either resolved or rejected. It may also be defined as a career which takes data from promise and further executes it successfully.
Parameters:
then() method takes two functions as parameters.
First function is executed if promise is resolved and a result is received.
Second function is executed if promise is rejected and an error is received. (It is optional and there is a better way to handle error using .catch() method
Syntax:

```
.then(function(result){
      //handle success
   }, function(error){
      //handle error
   })
```

Example: Promise Resolved

```
var promise = new Promise(function(resolve, reject) {
   resolve('Geeks For Geeks');
})

promise
   .then(function(successMessage) {
     //success handler function is invoked
      console.log(successMessage);
   }, function(errorMessage) {
      console.log(errorMessage);
   })
```

2. catch()
catch() is invoked when a promise is either rejected or some error has occurred in execution. It is used as an Error Handler whenever at any step there is a chance of getting an error.
Parameters:
catch() method takes one function as parameter.
Function to handle errors or promise rejections.(.catch() method internally calls .then(null, errorHandler), i.e. .catch() is just a shorthand for .then(null, errorHandler))

```
var promise = new Promise(function(resolve, reject) {
```

```
        reject('Promise Rejected')
})
promise
        .then(function(successMessage) {
                console.log(successMessage);
        })
        .catch(function(errorMessage) {
        //error handler function is invoked
                console.log(errorMessage);
        });
```

Applications
Promises are used for asynchronous handling of events.
Promises are used to handle asynchronous http requests.
The REST headers and parameters contain a wealth of information that can help you track down issues when you encounter them. HTTP Headers are an important part of the API request and response as they represent the meta-data associated with the API request and response. Headers carry information for:
Request and Response Body
Request Authorization
Response Caching
Response Cookies
https://www.edureka.co/blog/interview-questions/top-node-js-interview-questions-2016/
asked by client
database differences
endpoints for rest apis
parameter how use

buffer/stream difference
status code ask
404,400,401
postman header , get and post
why use
promises

pseudo code
angular js
hacker platform, 2h

frontend and backend proejct
upto 3pm time is there tomm

You can choose database of your choice as per your requirement. If you need to maintain strict data structure then choose relational db, else you can go for NO-SQL.
There are NPM packages for PostgreSQL, MySql and other db which are non-blocking. These db clients will not block the Node process while performing queries.
What is MongoDB?
MongoDB is an open-source document-oriented database used for high volume data storage. It falls under the classification of a NoSQL database. NoSQL tool means that it doesn't utilize the usual rows and columns. MongoDB uses BSON (document storage format) which is a binary style of JSON documents.
Features of MongoDB:
Multiple Servers: It can run over multiple servers.
Schema-less Database: It is a schema-less database.
Indexing: Any field in the document can be indexed.
Rich Object Model: It supports a rich object model.
What is RDBMS?
It stands for Relational Database Management System. It stores data in the form of related tables.
Features of RDBMS:
Gives a high level of information security.
It is quick and precise.
Provides facility primary key, to exceptionally distinguish the rows.
Difference between RDBMS and MongoDB:
RDBMS MongoDB
It is a relational database. It is a non-relational and document-oriented database.
Not suitable for hierarchical data storage. Suitable for hierarchical data storage.
It is vertically scalable i.e increasing RAM. It is horizontally scalable i.e we can add more servers.
It has a predefined schema. It has a dynamic schema.
It is quite vulnerable to SQL injection. It is not affected by SQL injection.
It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability). It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).
It is row-based. It is document-based.
It is slower in comparison with MongoDB. It is almost 100 times faster than RDBMS.
Supports complex joins. No support for complex joins.
It is column-based. It is field-based.
It does not provide JavaScript client for querying. It provides a JavaScript client for querying.
It supports SQL query language only. It supports JSON query language along with SQL.
To handle and manipulate streaming data like a video, a large file, etc., we need streams in Node. The streams module in Node.js manages all streams.
400 like password not found validation error or bad request
401 token expire or unauthorized
404 resource not found
429 too many requests
200 ok

204 no content
403 forbidden :client does not have access right to content
500 internal server error
502 bad gateway
503 service unavailable
504 gateway timeout
PATCH:
Patch request says that we would only send the data that we need to modify without modifying or effecting other parts of the data. Ex: if we need to update only the first name, we pass only the first name.
Please refer the below links for more information:
pseudocode.js is a JavaScript library that typesets pseudocode beautifully to HTML.

```
static async getUserList(
   req: Request,
   res: Response,
   next: NextFunction
 ): Promise<void> {
   try {
     const user = req['decoded'];
     const result = await UserService.getUserList(user, req.body);
     const response = new CustomResponse(res);
     response.setResponse({ result });
   }
   catch (e) {
     next(e);
   }
 }
MongoClient.connect(url).then((client) => {

   const db = client.db(database_name);

   database.insertDocument(db, { name: "Test",
      description: "Chill Out! Its just a test program!"},
      "test")
      .then((result) => {
         return database.findDocuments(db, "test");
      })
      .then((documents) => {
         console.log("Found Documents:\n", documents);

         return database.updateDocument(db, { name: "Test" },
               { description: "Updated Test" }, "test");
```

```
    })
    .then((result) => {
      console.log("Updated Documents Found:\n", result.result);

      return database.findDocuments(db, "test");
    })
    .then((docs) => {
      console.log("The Updated Documents are:\n", docs);

      return db.dropCollection("test");
    })
    .then((result) => {

      return client.close();
    })
    .catch((err) => alert(err));

})
.catch((err) => alert(err));
```

Now as compared to using the callbacks, our code looks a lot cleaner than before. As the .th

What are streams?

Streams are one of the fundamental concepts that power Node.js applications. They are data-handling method and are used to read or write input into output sequentially.

Streams are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

There are 4 types of streams in Node.js:

Writable: streams to which we can write data. For example, fs.createWriteStream() lets us write data to a file using streams.

Readable: streams from which data can be read. For example: fs.createReadStream() lets us read the contents of a file.

Duplex: streams that are both Readable and Writable. For example, net.Socket

Transform: streams that can modify or transform the data as it is written and read. For example, in the instance of file-compression, you can write compressed data and read decompressed data to and from a file.

Buffers in Node.js: The Buffer class in Node.js is used to perform operations on raw binary data. Generally, Buffer refers to the particular memory location in memory. Buffer and array have some similarities, but the difference is array can be any type, and it can be resizable. Buffers only deal with binary data, and it can not be resizable. Each integer in a buffer represents a byte. console.log() function is used to print the Buffer instance.

Methods to perform the operations on Buffer:

1	Buffer.alloc(size)	It creates a buffer and allocates size to it.
2	Buffer.from(initialization)	It initializes the buffer with given data.

3	Buffer.write(data)	It writes the data on the buffer.
4	toString()	It read data from the buffer and returned it.
5	Buffer.isBuffer(object)	It checks whether the object is a buffer or not.
6	Buffer.length	It returns the length of the buffer.

api way to write

```
router.route('/setPassword')
  .post(validate(authValidation.setPassword), AuthController.setPassword);
```

The Event Loop in Node.js

The Event Loop is composed of the following six phases, which are repeated for as long as the application still has code that needs to be executed:

Timers

I/O Callbacks

Waiting / Preparation

I/O Polling

setImmediate() callbacks

Close events

The Event Loop starts at the moment Node.js begins to execute your index.js file, or any other application entry point.

These six phases create one cycle, or loop, which is known as a tick. A Node.js process exits when there is no more pending work in the Event Loop, or when process.exit() is called manually. A program only runs for as long as there are tasks queued in the Event Loop, or present on the call stack.

Know how to build REST API with Node.js from scratch

Since the invention of WWW, various web technologies like RPC or SOAP were used to create and implement web services. But these technologies used heavy definitions for handling any communication task. Thus, REST was developed which helped in reducing the complexities and provided an architectural style in order to design the network-based application. Since Node.js technology is revolutionizing the server for the front-end developers, in this article I will be demonstrating the process of Building REST API with Node.js from scratch.

Below are the topics that I will be covering in this article:

What is REST API?

Principles of REST

Practical Demonstration: Building a REST API with Node.js

What is REST API?

REST or RESTful stands for REpresentational State Transfer. It is an architectural style as well as an approach for communications purposes that is often used in various web services development. In simpler terms, it is an application program interface (API) that makes use of the HTTP requests to GET, PUT, POST and DELETE the data over WWW.

REST architectural style helps in leveraging the lesser use of bandwidth which makes an application more suitable for the internet. It is often regarded as the “language of the internet”. It is completely based on the resources where each and every component is regarded as a component and a single resource is accessible through a common interface using the standard HTTP method.

To understand better, let's dive a little deeper and see how exactly does a REST API work. Basically, the REST API breaks down a transaction in order to create small modules. Now, each of these modules is used to address a specific part of the transaction. This approach provides more flexibility but requires a lot of effort to be built from the very scratch.
The main functions used in any REST-based architecture are:
GET − Provides read-only access to a resource.
PUT − Creates a new resource.
DELETE − Removes a resource.
POST − Updates an existing resource or creates a new resource.
00olBut all who claims cannot be referred to as RESTful API. In order to be regarded as a RESTful API, your application must satisfy certain constraints or principles. In the next section of this article on Building a REST API using Node.js, I will be talking about these principles in detail.
Principles of REST
Well, there are six ground principles laid down by Dr. Fielding who was the one to define the REST API design in 2000. Below are the six guiding principles of REST:
Stateless
Requests sent from a client to the server contains all the necessary information that is required to completely understand it. It can be a part of the URI, query-string parameters, body, or even headers. The URI is used for uniquely identifying the resource and the body holds the state of the requesting resource. Once the processing is done by the server, an appropriate response is sent back to the client through headers, status or response body.
Client-Server
It has a uniform interface that separates the clients from the servers. Separating the concerns helps in improving the user interface's portability across multiple platforms as well as enhance the scalability of the server components.
Uniform Interface
To obtain the uniformity throughout the application, REST has defined four interface constraints which are:
Resource identification
Resource Manipulation using representations
Self-descriptive messages
Hypermedia as the engine of application state
Cacheable
In order to provide a better performance, the applications are often made cacheable. It is done by labeling the response from the server as cacheable or non-cacheable either implicitly or explicitly. If the response is defined as cacheable, then the client cache can reuse the response data for equivalent responses in the future. It also helps in preventing the reuse of the stale data.
Layered system
The layered system architecture allows an application to be more stable by limiting component behavior. This architecture enables load balancing and provides shared caches for promoting scalability. The layered architecture also helps in enhancing the application's security as components in each layer cannot interact beyond the next immediate layer they are in.

Code on demand
Code on Demand is an optional constraint and is used the least. It permits a clients code or applets to be downloaded and extended via the interface to be used within the application. In essence, it simplifies the clients by creating a smart application which doesn't rely on its own code structure.
1. What is a first class function in Javascript?
When functions can be treated like any other variable then those functions are first-class functions. There are many other programming languages, for example, scala, Haskell, etc which follow this including JS. Now because of this function can be passed as a param to another function(callback) or a function can return another function(higher-order function). map() and filter() are higher-order functions that are popularly used.
2. What is Node.js and how it works?
Node.js is a virtual machine that uses JavaScript as its scripting language and runs Chrome's V8 JavaScript engine. Basically, Node.js is based on an event-driven architecture where I/O runs asynchronously making it lightweight and efficient. It is being used in developing desktop applications as well with a popular framework called electron as it provides API to access OS-level features such as file system, network, etc.
3. How do you manage packages in your node.js project?
It can be managed by a number of package installers and their configuration file accordingly. Out of them mostly use npm or yarn. Both provide almost all libraries of javascript with extended features of controlling environment-specific configurations. To maintain versions of libs being installed in a project we use package.json and package-lock.json so that there is no issue in porting that app to a different environment.
You can download a PDF version of Node Js Interview Questions.
Download PDF
4. How is Node.js better than other frameworks most popularly used?
Node.js provides simplicity in development because of its non-blocking I/O and even-based model results in short response time and concurrent processing, unlike other frameworks where developers have to use thread management.

It runs on a chrome v8 engine which is written in c++ and is highly performant with constant improvement.

Also since we will use Javascript in both the frontend and backend the development will be much faster.

And at last, there are ample libraries so that we don't need to reinvent the wheel.
5. Explain the steps how "Control Flow" controls the functions calls?
Control the order of execution
Collect data
Limit concurrency
Call the following step in the program.
6. What are some commonly used timing features of Node.js?
setTimeout/clearTimeout – This is used to implement delays in code execution.

setInterval/clearInterval – This is used to run a code block multiple times.
setImmediate/clearImmediate – Any function passed as the setImmediate() argument is a callback that's executed in the next iteration of the event loop.
process.nextTick – Any function passed as the setImmediate() argument is a callback that's executed in the next iteration of the event loop.
7. What are the advantages of using promises instead of callbacks?
The main advantage of using promise is you get an object to decide the action that needs to be taken after the async task completes. This gives more manageable code and avoids callback hell.
8. What is fork in node JS?
A fork in general is used to spawn child processes. In node it is used to create a new instance of v8 engine to run multiple workers to execute the code.
9. Why is Node.js single-threaded?
Node.js was created explicitly as an experiment in async processing. This was to try a new theory of doing async processing on a single thread over the existing thread-based implementation of scaling via different frameworks.
10. How do you create a simple server in Node.js that returns Hello World?

```
var http = require("http");
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(3000);
```

11. How many types of API functions are there in Node.js?
There are two types of API functions:
Asynchronous, non-blocking functions - mostly I/O operations which can be fork out of the main loop.

Synchronous, blocking functions - mostly operations that influence the process running in the main loop.
12. What is REPL?
PL in Node.js stands for Read, Eval, Print, and Loop, which further means evaluating code on the go.
13. List down the two arguments that async.queue takes as input?
Task Function
Concurrency Value
14. What is the purpose of module.exports?
This is used to expose functions of a particular module or file to be used elsewhere in the project. This can be used to encapsulate all similar functions in a file which further improves the project structure. For example, you have a file for all utils functions with util to get solutions in a different programming language of a problem statement.

```
const getSolutionInJavaScript = async ({
 problem_id
}) => {
...
};
const getSolutionInPython = async ({
```

```
 problem_id
}) => {
...
};
module.exports = { getSolutionInJavaScript, getSolutionInPython }
```

Thus using module.exports we can use these functions in some other file:

```
const { getSolutionInJavaScript, getSolutionInPython} = require("./utils")
```

15. What tools can be used to assure consistent code style?

ESLint can be used with any IDE to ensure a consistent coding style which further helps in maintaining the codebase.

Intermediate Node.js Interview Questions

16. What do you understand by callback hell?

```
async_A(function(){
  async_B(function(){
     async_C(function(){
        async_D(function(){
        ....
        });
     });
  });
});
```

For the above example, we are passing callback functions and it makes the code unreadable and not maintainable, thus we should change the async logic to avoid this.

17. What is an event-loop in Node JS?

Whatever that is async is managed by event-loop using a queue and listener. We can get the idea using the following diagram:

done.....

Node.js Event Loop

So when an async function needs to be executed(or I/O) the main thread sends it to a different thread allowing v8 to keep executing the main code. Event loop involves different phases with specific tasks such as timers, pending callbacks, idle or prepare, poll, check, close callbacks with different FIFO queues. Also in between iterations it checks for async I/O or timers and shuts down cleanly if there aren't any.

18. If Node.js is single threaded then how does it handle concurrency?

The main loop is single-threaded and all async calls are managed by libuv library.

For example:

```
const crypto = require("crypto");
const start = Date.now();
function logHashTime() {
 crypto.pbkdf2("a", "b", 100000, 512, "sha512", () => {
console.log("Hash: ", Date.now() - start);
 });
}
```

```
logHashTime();
logHashTime();
logHashTime();
logHashTime();
```

This gives the output:

```
Hash: 1213
Hash: 1225
Hash: 1212
Hash: 1222
```

This is because libuv sets up a thread pool to handle such concurrency. How many threads will be there in the thread pool depends upon the number of cores but you can override this.

19. Differentiate between process.nextTick() and setImmediate()?

Both can be used to switch to an asynchronous mode of operation by listener functions.

process.nextTick() sets the callback to execute but setImmediate pushes the callback in the queue to be executed. So the event loop runs in the following manner

timers–>pending callbacks–>idle,prepare–>connections(poll,data,etc)–>check–>close callbacks

In this process.nextTick() method adds the callback function to the start of the next event queue and setImmediate() method to place the function in the check phase of the next event queue.

20. How does Node.js overcome the problem of blocking of I/O operations?

Since the node has an event loop that can be used to handle all the I/O operations in an asynchronous manner without blocking the main function.

So for example, if some network call needs to happen it will be scheduled in the event loop instead of the main thread(single thread). And if there are multiple such I/O calls each one will be queued accordingly to be executed separately(other than the main thread).

Thus even though we have single-threaded JS, I/O ops are handled in a nonblocking way.

21. How can we use async await in node.js?

Here is an example of using async-await pattern:

```
// this code is to retry with exponential backoff
function wait (timeout) {
 return new Promise((resolve) => {
setTimeout(() => {
  resolve()
}, timeout);
 });
}
async function requestWithRetry (url) {
 const MAX_RETRIES = 10;
 for (let i = 0; i <= MAX_RETRIES; i++) {
try {
  return await request(url);
} catch (err) {
  const timeout = Math.pow(2, i);
```

```
  console.log('Waiting', timeout, 'ms');
  await wait(timeout);
  console.log('Retrying', err.message, i);
}
 }
}
```

22. What is node.js streams?

Streams are instances of EventEmitter which can be used to work with streaming data in Node.js. They can be used for handling and manipulating streaming large files(videos, mp3, etc) over the network. They use buffers as their temporary storage.

There are mainly four types of the stream:

Writable: streams to which data can be written (for example, fs.createWriteStream()).

Readable: streams from which data can be read (for example, fs.createReadStream()).

Duplex: streams that are both Readable and Writable (for example, net.Socket).

Transform: Duplex streams that can modify or transform the data as it is written and read (for example, zlib.createDeflate()).

23. What are node.js buffers?

In general, buffers is a temporary memory that is mainly used by stream to hold on to some data until consumed. Buffers are introduced with additional use cases than JavaScript's Unit8Array and are mainly used to represent a fixed-length sequence of bytes. This also supports legacy encodings like ASCII, utf-8, etc. It is a fixed(non-resizable) allocated memory outside the v8.

24. What is middleware?

Middleware comes in between your request and business logic. It is mainly used to capture logs and enable rate limit, routing, authentication, basically whatever that is not a part of business logic. There are third-party middleware also such as body-parser and you can write your own middleware for a specific use case.

25. Explain what a Reactor Pattern in Node.js?

Reactor pattern again a pattern for nonblocking I/O operations. But in general, this is used in any event-driven architecture.

There are two components in this: 1. Reactor 2. Handler.

Reactor: Its job is to dispatch the I/O event to appropriate handlers

Handler: Its job is to actually work on those events

26. Why should you separate Express app and server?

The server is responsible for initializing the routes, middleware, and other application logic whereas the app has all the business logic which will be served by the routes initiated by the server. This ensures that the business logic is encapsulated and decoupled from the application logic which makes the project more readable and maintainable.

27. For Node.js, why Google uses V8 engine?

Well, are there any other options available? Yes, of course, we have Spidermonkey from Firefox, Chakra from Edge but Google's v8 is the most evolved(since it's open-source so there's a huge community helping in developing features and fixing bugs) and fastest(since it's written in c++) we got till now as a JavaScript and WebAssembly engine. And it is portable to almost every machine known.

28. Describe the exit codes of Node.js?
Exit codes give us an idea of how a process got terminated/the reason behind termination.
A few of them are:
Uncaught fatal exception - (code - 1) - There has been an exception that is not handled
Unused - (code - 2) - This is reserved by bash
Fatal Error - (code - 5) - There has been an error in V8 with stderr output of the description
Internal Exception handler Run-time failure - (code - 7) - There has been an exception when bootstrapping function was called
Internal JavaScript Evaluation Failure - (code - 4) - There has been an exception when the bootstrapping process failed to return function value when evaluated.
29. Explain the concept of stub in Node.js?
Stubs are used in writing tests which are an important part of development. It replaces the whole function which is getting tested.
This helps in scenarios where we need to test:
External calls which make tests slow and difficult to write (e.g HTTP calls/ DB calls)
Triggering different outcomes for a piece of code (e.g. what happens if an error is thrown/ if it passes)
For example, this is the function:

```
const request = require('request');
const getPhotosByAlbumId = (id) => {
const requestUrl = `https://jsonplaceholder.typicode.com/albums/${id}/photos?_limit=3`;
return new Promise((resolve, reject) => {
  request.get(requestUrl, (err, res, body) => {
    if (err) {
      return reject(err);
    }
    resolve(JSON.parse(body));
  });
});
};
module.exports = getPhotosByAlbumId;
```

To test this function this is the stub

```
const expect = require('chai').expect;
const request = require('request');
const sinon = require('sinon');
const getPhotosByAlbumId = require('./index');
describe('with Stub: getPhotosByAlbumId', () => {
before(() => {
  sinon.stub(request, 'get')
    .yields(null, null, JSON.stringify([
      {
        "albumId": 1,
        "id": 1,
```

```
            "title": "A real photo 1",
            "url": "https://via.placeholder.com/600/92c952",
            "thumbnailUrl": "https://via.placeholder.com/150/92c952"
         },
         {
            "albumId": 1,
            "id": 2,
            "title": "A real photo 2",
            "url": "https://via.placeholder.com/600/771796",
            "thumbnailUrl": "https://via.placeholder.com/150/771796"
         },
         {
            "albumId": 1,
            "id": 3,
            "title": "A real photo 3",
            "url": "https://via.placeholder.com/600/24f355",
            "thumbnailUrl": "https://via.placeholder.com/150/24f355"
         }
      ]));
});
after(() => {
   request.get.restore();
});
it('should getPhotosByAlbumId', (done) => {
   getPhotosByAlbumId(1).then((photos) => {
      expect(photos.length).to.equal(3);
      photos.forEach(photo => {
         expect(photo).to.have.property('id');
         expect(photo).to.have.property('title');
         expect(photo).to.have.property('url');
      });
      done();
   });
});
});
```

Advanced Node.js Interview Questions

30. What is an Event Emitter in Node.js?

EventEmitter is a Node.js class that includes all the objects that are basically capable of emitting events. This can be done by attaching named events that are emitted by the object using an eventEmitter.on() function. Thus whenever this object throws an even the attached functions are invoked synchronously.

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
```

```
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
 console.log('an event occurred!');
});
myEmitter.emit('event');
```

31. Enhancing Node.js performance through clustering.

Node.js applications run on a single processor, which means that by default they don’t take advantage of a multiple-core system. Cluster mode is used to start up multiple node.js processes thereby having multiple instances of the event loop. When we start using cluster in a nodejs app behind the scene multiple node.js processes are created but there is also a parent process called the cluster manager which is responsible for monitoring the health of the individual instances of our application.

Clustering in Node.js

32. What is a thread pool and which library handles it in Node.js

The Thread pool is handled by the libuv library. libuv is a multi-platform C library that provides support for asynchronous I/O-based operations such as file systems, networking, and concurrency.

Thread Pool

33. What is WASI and why is it being introduced?

Web assembly provides an implementation of WebAssembly System Interface specification through WASI API in node.js implemented using WASI class. The introduction of WASI was done by keeping in mind its possible to use the underlying operating system via a collection of POSIX-like functions thus further enabling the application to use resources more efficiently and features that require system-level access.

34. How are worker threads different from clusters?

Cluster:

There is one process on each CPU with an IPC to communicate.

In case we want to have multiple servers accepting HTTP requests via a single port, clusters can be helpful.

The processes are spawned in each CPU thus will have separate memory and node instance which further will lead to memory issues.

Worker threads:

There is only one process in total with multiple threads.

Each thread has one Node instance (one event loop, one JS engine) with most of the APIs accessible.

Shares memory with other threads (e.g. SharedArrayBuffer)

This can be used for CPU-intensive tasks like processing data or accessing the file system since NodeJS is single-threaded, synchronous tasks can be made more efficient leveraging the worker's threads.

35. How to measure the duration of async operations?

Performance API provides us with tools to figure out the necessary performance metrics. A simple example would be using async_hooks and perf_hooks

```
'use strict';
const async_hooks = require('async_hooks');
const {
 performance,
```

```
 PerformanceObserver
} = require('perf_hooks');
const set = new Set();
const hook = async_hooks.createHook({
 init(id, type) {
if (type === 'Timeout') {
  performance.mark(`Timeout-${id}-Init`);
  set.add(id);
}
 },
 destroy(id) {
if (set.has(id)) {
  set.delete(id);
  performance.mark(`Timeout-${id}-Destroy`);
  performance.measure(`Timeout-${id}`,
                `Timeout-${id}-Init`,
                `Timeout-${id}-Destroy`);
}
 }
});
hook.enable();
const obs = new PerformanceObserver((list, observer) => {
 console.log(list.getEntries()[0]);
 performance.clearMarks();
 observer.disconnect();
});
obs.observe({ entryTypes: ['measure'], buffered: true });
setTimeout(() => {}, 1000);
```

This would give us the exact time it took to execute the callback.

36. How to measure the performance of async operations?

Performance API provides us with tools to figure out the necessary performance metrics.

A simple example would be:

```
const { PerformanceObserver, performance } = require('perf_hooks');
const obs = new PerformanceObserver((items) => {
 console.log(items.getEntries()[0].duration);
 performance.clearMarks();
});
obs.observe({ entryTypes: ['measure'] });
performance.measure('Start to Now');
performance.mark('A');
doSomeLongRunningProcess(() => {
 performance.measure('A to Now', 'A');
```

```
 performance.mark('B');
 performance.measure('A to B', 'A', 'B');
});
```

Node.Js Interview MCQs

1.

Which of the following statements is valid to import a module in file?

var fs = require("fs");

var fs = import("fs");

package fs;

import fs;

2.

What is the fullform of NPM?

New Package Manager

Node Package Manager

Node Project Manager

3.

Is Node multithreaded?

True

False

4.

How to make node modules available externally?

module.expose

module.spread

module.export

5.

Which of the following are Node.js stream types?

Principles of REST

Well, there are six ground principles laid down by Dr. Fielding who was the one to define the REST API design in 2000. Below are the six guiding principles of REST:

Stateless

Requests sent from a client to the server contains all the necessary information that is required to completely understand it. It can be a part of the URI, query-string parameters, body, or even headers. The URI is used for uniquely identifying the resource and the body holds the state of the requesting resource. Once the processing is done by the server, an appropriate response is sent back to the client through headers, status or response body.

Client-Server

It has a uniform interface that separates the clients from the servers. Separating the concerns helps in improving the user interface's portability across multiple platforms as well as enhance the scalability of the server components.

Uniform Interface

To obtain the uniformity throughout the application, REST has defined four interface constraints which are:

Resource identification

Resource Manipulation using representations

Self-descriptive messages

Hypermedia as the engine of application state

Cacheable

In order to provide a better performance, the applications are often made cacheable. It is done by labeling the response from the server as cacheable or non-cacheable either implicitly or explicitly. If the response is defined as cacheable, then the client cache can reuse the response data for equivalent responses in the future. It also helps in preventing the reuse of the stale data.

Layered system

The layered system architecture allows an application to be more stable by limiting component behavior. This architecture enables load balancing and provides shared caches for promoting scalability. The layered architecture also helps in enhancing the application's security as components in each layer cannot interact beyond the next immediate layer they are in.

Code on demand

Code on Demand is an optional constraint and is used the least. It permits a clients code or applets to be downloaded and extended via the interface to be used within the application. In essence, it simplifies the clients by creating a smart application which doesn't rely on its own code structure.

Beginner Node.js Interview Questions

1. What is a first class function in Javascript?

When functions can be treated like any other variable then those functions are first-class functions. There are many other programming languages, for example, scala, Haskell, etc which follow this including JS. Now because of this function can be passed as a param to another function(callback) or a function can return another function(higher-order function). map() and filte r() are higher-order functions that are popularly used.

2. What is Node.js and how it works?

Node.js is a virtual machine that uses JavaScript as its scripting language and runs Chrome's V8 JavaScript engine. Basically, Node.js is based on an event-driven architecture where I/O runs asynchronously making it lightweight and efficient. It is being used in developing desktop applications as well with a popular framework called electron as it provides API to access OS-level features such as file system, network, etc.

3. How do you manage packages in your node.js project?

It can be managed by a number of package installers and their configuration file accordingly. Out of them mostly use npm or yarn. Both provide almost all libraries of javascript with extended features of controlling environment-specific configurations. To maintain versions of libs being installed in a project we use package.json and package-lock.json so that there is no issue in porting that app to a different environment.

You can download a PDF version of Node Js Interview Questions.

Download PDF

4. How is Node.js better than other frameworks most popularly used?

Node.js provides simplicity in development because of its non-blocking I/O and even-based model results in short response time and concurrent processing, unlike other frameworks where developers have to use thread management.

It runs on a chrome v8 engine which is written in c++ and is highly performant with constant improvement.

Also since we will use Javascript in both the frontend and backend the development will be much faster.

And at last, there are ample libraries so that we don't need to reinvent the wheel.

5. Explain the steps how "Control Flow" controls the functions calls?

Control the order of execution
Collect data
Limit concurrency
Call the following step in the program.

6. What are some commonly used timing features of Node.js?

setTimeout/clearTimeout – This is used to implement delays in code execution.
setInterval/clearInterval – This is used to run a code block multiple times.
setImmediate/clearImmediate – Any function passed as the setImmediate() argument is a callback that's executed in the next iteration of the event loop.
process.nextTick – Any function passed as the setImmediate() argument is a callback that's executed in the next iteration of the event loop.

7. What are the advantages of using promises instead of callbacks?

The main advantage of using promise is you get an object to decide the action that needs to be taken after the async task completes. This gives more manageable code and avoids callback hell.

8. What is fork in node JS?

A fork in general is used to spawn child processes. In node it is used to create a new instance of v8 engine to run multiple workers to execute the code.

9. Why is Node.js single-threaded?

Node.js was created explicitly as an experiment in async processing. This was to try a new theory of doing async processing on a single thread over the existing thread-based implementation of scaling via different frameworks.

10. How do you create a simple server in Node.js that returns Hello World?

```
var http = require("http");
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(3000);
```

11. How many types of API functions are there in Node.js?

There are two types of API functions:
Asynchronous, non-blocking functions - mostly I/O operations which can be fork out of the main loop.

Synchronous, blocking functions - mostly operations that influence the process running in the main loop.

12. What is REPL?

PL in Node.js stands for Read, Eval, Print, and Loop, which further means evaluating code on the go.

13. List down the two arguments that async.queue takes as input?

Task Function

Concurrency Value

14. What is the purpose of module.exports?

This is used to expose functions of a particular module or file to be used elsewhere in the project. This can be used to encapsulate all similar functions in a file which further improves the project structure. For example, you have a file for all utils functions with util to get solutions in a different programming language of a problem statement.

```
const getSolutionInJavaScript = async ({
 problem_id
}) => {
...
};
const getSolutionInPython = async ({
 problem_id
}) => {
...
};
module.exports = { getSolutionInJavaScript, getSolutionInPython }
```

Thus using module.exports we can use these functions in some other file:

```
const { getSolutionInJavaScript, getSolutionInPython} = require("./utils")
```

15. What tools can be used to assure consistent code style?

ESLint can be used with any IDE to ensure a consistent coding style which further helps in maintaining the codebase.

Intermediate Node.js Interview Questions

16. What do you understand by callback hell?

```
async_A(function(){
  async_B(function(){
     async_C(function(){
        async_D(function(){
        ....
        });
     });
  });
});
```

For the above example, we are passing callback functions and it makes the code unreadable and not maintainable, thus we should change the async logic to avoid this.

17. What is an event-loop in Node JS?

Whatever that is async is managed by event-loop using a queue and listener. We can get the idea using the following diagram:
Node.js Event Loop
So when an async function needs to be executed(or I/O) the main thread sends it to a different thread allowing v8 to keep executing the main code. Event loop involves different phases with specific tasks such as timers, pending callbacks, idle or prepare, poll, check, close callbacks with different FIFO queues. Also in between iterations it checks for async I/O or timers and shuts down cleanly if there aren't any.
18. If Node.js is single threaded then how does it handle concurrency?
The main loop is single-threaded and all async calls are managed by libuv library.
For example:

```
const crypto = require("crypto");
const start = Date.now();
function logHashTime() {
 crypto.pbkdf2("a", "b", 100000, 512, "sha512", () => {
console.log("Hash: ", Date.now() - start);
 });
}
logHashTime();
logHashTime();
logHashTime();
logHashTime();
```

This gives the output:

```
Hash: 1213
Hash: 1225
Hash: 1212
Hash: 1222
```

This is because libuv sets up a thread pool to handle such concurrency. How many threads will be there in the thread pool depends upon the number of cores but you can override this.
19. Differentiate between process.nextTick() and setImmediate()?
Both can be used to switch to an asynchronous mode of operation by listener functions.
process.nextTick() sets the callback to execute but setImmediate pushes the callback in the queue to be executed. So the event loop runs in the following manner
timers–>pending callbacks–>idle,prepare–>connections(poll,data,etc)–>check–>close callbacks
In this process.nextTick() method adds the callback function to the start of the next event queue and setImmediate() method to place the function in the check phase of the next event queue.
20. How does Node.js overcome the problem of blocking of I/O operations?
Since the node has an event loop that can be used to handle all the I/O operations in an asynchronous manner without blocking the main function.
So for example, if some network call needs to happen it will be scheduled in the event loop instead of the main thread(single thread). And if there are multiple such I/O calls each one will be queued accordingly to be executed separately(other than the main thread).
Thus even though we have single-threaded JS, I/O ops are handled in a nonblocking way.

21. How can we use async await in node.js?
Here is an example of using async-await pattern:

```
// this code is to retry with exponential backoff
function wait (timeout) {
 return new Promise((resolve) => {
setTimeout(() => {
  resolve()
}, timeout);
 });
}
async function requestWithRetry (url) {
 const MAX_RETRIES = 10;
 for (let i = 0; i <= MAX_RETRIES; i++) {
try {
  return await request(url);
} catch (err) {
  const timeout = Math.pow(2, i);
  console.log('Waiting', timeout, 'ms');
  await wait(timeout);
  console.log('Retrying', err.message, i);
}
 }
}
```

22. What is node.js streams?
Streams are instances of EventEmitter which can be used to work with streaming data in Node.js. They can be used for handling and manipulating streaming large files(videos, mp3, etc) over the network. They use buffers as their temporary storage.
There are mainly four types of the stream:
Writable: streams to which data can be written (for example, fs.createWriteStream()).
Readable: streams from which data can be read (for example, fs.createReadStream()).
Duplex: streams that are both Readable and Writable (for example, net.Socket).
Transform: Duplex streams that can modify or transform the data as it is written and read (for example, zlib.createDeflate()).
23. What are node.js buffers?
In general, buffers is a temporary memory that is mainly used by stream to hold on to some data until consumed. Buffers are introduced with additional use cases than JavaScript's Unit8Array and are mainly used to represent a fixed-length sequence of bytes. This also supports legacy encodings like ASCII, utf-8, etc. It is a fixed(non-resizable) allocated memory outside the v8.
24. What is middleware?
Middleware comes in between your request and business logic. It is mainly used to capture logs and enable rate limit, routing, authentication, basically whatever that is not a part of business logic. There

are third-party middleware also such as body-parser and you can write your own middleware for a specific use case.

25. Explain what a Reactor Pattern in Node.js?

Reactor pattern again a pattern for nonblocking I/O operations. But in general, this is used in any event-driven architecture.

There are two components in this: 1. Reactor 2. Handler.

Reactor: Its job is to dispatch the I/O event to appropriate handlers

Handler: Its job is to actually work on those events

26. Why should you separate Express app and server?

The server is responsible for initializing the routes, middleware, and other application logic whereas the app has all the business logic which will be served by the routes initiated by the server. This ensures that the business logic is encapsulated and decoupled from the application logic which makes the project more readable and maintainable.

27. For Node.js, why Google uses V8 engine?

Well, are there any other options available? Yes, of course, we have Spidermonkey from Firefox, Chakra from Edge but Google's v8 is the most evolved(since it's open-source so there's a huge community helping in developing features and fixing bugs) and fastest(since it's written in c++) we got till now as a JavaScript and WebAssembly engine. And it is portable to almost every machine known.

28. Describe the exit codes of Node.js?

Exit codes give us an idea of how a process got terminated/the reason behind termination.

A few of them are:

Uncaught fatal exception - (code - 1) - There has been an exception that is not handled

Unused - (code - 2) - This is reserved by bash

Fatal Error - (code - 5) - There has been an error in V8 with stderr output of the description

Internal Exception handler Run-time failure - (code - 7) - There has been an exception when bootstrapping function was called

Internal JavaScript Evaluation Failure - (code - 4) - There has been an exception when the bootstrapping process failed to return function value when evaluated.

29. Explain the concept of stub in Node.js?

Stubs are used in writing tests which are an important part of development. It replaces the whole function which is getting tested.

This helps in scenarios where we need to test:

External calls which make tests slow and difficult to write (e.g HTTP calls/ DB calls)

Triggering different outcomes for a piece of code (e.g. what happens if an error is thrown/ if it passes)

For example, this is the function:

```
const request = require('request');
const getPhotosByAlbumId = (id) => {
const requestUrl = `https://jsonplaceholder.typicode.com/albums/${id}/photos?_limit=3`;
return new Promise((resolve, reject) => {
  request.get(requestUrl, (err, res, body) => {
    if (err) {
      return reject(err);
```

```
      }
      resolve(JSON.parse(body));
   });
});
};
module.exports = getPhotosByAlbumId;
To test this function this is the stub
const expect = require('chai').expect;
const request = require('request');
const sinon = require('sinon');
const getPhotosByAlbumId = require('./index');
describe('with Stub: getPhotosByAlbumId', () => {
before(() => {
   sinon.stub(request, 'get')
      .yields(null, null, JSON.stringify([
         {
            "albumId": 1,
            "id": 1,
            "title": "A real photo 1",
            "url": "https://via.placeholder.com/600/92c952",
            "thumbnailUrl": "https://via.placeholder.com/150/92c952"
         },
         {
            "albumId": 1,
            "id": 2,
            "title": "A real photo 2",
            "url": "https://via.placeholder.com/600/771796",
            "thumbnailUrl": "https://via.placeholder.com/150/771796"
         },
         {
            "albumId": 1,
            "id": 3,
            "title": "A real photo 3",
            "url": "https://via.placeholder.com/600/24f355",
            "thumbnailUrl": "https://via.placeholder.com/150/24f355"
         }
      ]));
});
after(() => {
   request.get.restore();
});
it('should getPhotosByAlbumId', (done) => {
```

```
  getPhotosByAlbumId(1).then((photos) => {
    expect(photos.length).to.equal(3);
    photos.forEach(photo => {
      expect(photo).to.have.property('id');
      expect(photo).to.have.property('title');
      expect(photo).to.have.property('url');
    });
    done();
  });
});
});
```

Advanced Node.js Interview Questions

30. What is an Event Emitter in Node.js?

EventEmitter is a Node.js class that includes all the objects that are basically capable of emitting events. This can be done by attaching named events that are emitted by the object using an eventEmitter.on() function. Thus whenever this object throws an even the attached functions are invoked synchronously.

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
 console.log('an event occurred!');
});
myEmitter.emit('event');
```

31. Enhancing Node.js performance through clustering.

Node.js applications run on a single processor, which means that by default they don’t take advantage of a multiple-core system. Cluster mode is used to start up multiple node.js processes thereby having multiple instances of the event loop. When we start using cluster in a nodejs app behind the scene multiple node.js processes are created but there is also a parent process called the cluster manager which is responsible for monitoring the health of the individual instances of our application.

Clustering in Node.js

32. What is a thread pool and which library handles it in Node.js

The Thread pool is handled by the libuv library. libuv is a multi-platform C library that provides support for asynchronous I/O-based operations such as file systems, networking, and concurrency.

Thread Pool

33. What is WASI and why is it being introduced?

Web assembly provides an implementation of WebAssembly System Interface specification through WASI API in node.js implemented using WASI class. The introduction of WASI was done by keeping in mind its possible to use the underlying operating system via a collection of POSIX-like functions thus further enabling the application to use resources more efficiently and features that require system-level access.

34. How are worker threads different from clusters?

Cluster:

There is one process on each CPU with an IPC to communicate.
In case we want to have multiple servers accepting HTTP requests via a single port, clusters can be helpful.
The processes are spawned in each CPU thus will have separate memory and node instance which further will lead to memory issues.
Worker threads:
There is only one process in total with multiple threads.
Each thread has one Node instance (one event loop, one JS engine) with most of the APIs accessible.
Shares memory with other threads (e.g. SharedArrayBuffer)
This can be used for CPU-intensive tasks like processing data or accessing the file system since NodeJS is single-threaded, synchronous tasks can be made more efficient leveraging the worker's threads.
35. How to measure the duration of async operations?
Performance API provides us with tools to figure out the necessary performance metrics. A simple example would be using async_hooks and perf_hooks

```
'use strict';
const async_hooks = require('async_hooks');
const {
 performance,
 PerformanceObserver
} = require('perf_hooks');
const set = new Set();
const hook = async_hooks.createHook({
 init(id, type) {
if (type === 'Timeout') {
  performance.mark(`Timeout-${id}-Init`);
  set.add(id);
}
 },
 destroy(id) {
if (set.has(id)) {
  set.delete(id);
  performance.mark(`Timeout-${id}-Destroy`);
  performance.measure(`Timeout-${id}`,
                      `Timeout-${id}-Init`,
                      `Timeout-${id}-Destroy`);
}
 }
});
hook.enable();
const obs = new PerformanceObserver((list, observer) => {
 console.log(list.getEntries()[0]);
 performance.clearMarks();
```

```
 observer.disconnect();
});
obs.observe({ entryTypes: ['measure'], buffered: true });
setTimeout(() => {}, 1000);
```

This would give us the exact time it took to execute the callback.

36. How to measure the performance of async operations?

Performance API provides us with tools to figure out the necessary performance metrics.

A simple example would be:

```
const { PerformanceObserver, performance } = require('perf_hooks');
const obs = new PerformanceObserver((items) => {
 console.log(items.getEntries()[0].duration);
 performance.clearMarks();
});
obs.observe({ entryTypes: ['measure'] });
performance.measure('Start to Now');
performance.mark('A');
doSomeLongRunningProcess(() => {
 performance.measure('A to Now', 'A');
 performance.mark('B');
 performance.measure('A to B', 'A', 'B');
});
```

Features of GraphQL

Here are important features of GraphQL:

It is statically typed, so you do not need to define variable before using it.

GraphQL can decouple frontend from backend.

No over or under fetching of data.

It is language and HTTP agnostic.

Documentation of GraqphQL comes with no extra cost.

It helps you to save bandwidth.

Object–relational mapping (ORM, O/RM, and O/R mapping tool) in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.

for uploading a big file

Multer should work fine. It'll stream the data as it comes in over HTTP to a file on disk. It won't lock up your server. Then in your endpoint you have access to that file location and can do whatever you need with it.

Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files. It is written on top of busboy for maximum efficiency.