1. 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.

2. What are the benefits of using Node.js?

From a web server development perspective Node has a number of benefits:

- Great performance! Node was designed to optimize throughput and scalability in web applications and is a good solution for many common web-development problems (e.g. real-time web applications).
- Code is written in "plain old JavaScript", which means that less time is spent dealing with "context shift" between languages when you're writing both client-side and server-side code.
- JavaScript is a relatively new programming languages and benefits from improvements in language design when compared to other traditional web-server languages (e.g. Python, PHP, etc.) Many other new and pouplar languages compile/convert into JavaScript so you can use TypeScript, CoffeeScript, ClojureScript, Scala, LiveScript, etc.
- The node package manager (NPM) provides access to hundres of thousands of resuable packages. It also has best-in-class dependency resolution and can also be used to automate most of the build toolchain.
- Node.js is portable. It is available on Microsoft Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS, and NonStop OS. Furthermore, it is well-supported by many web hosting providers, that often provide specific infrastrucutre and documentation for hosting Node sites.
- It has a very active third party ecosystem and developer community, with lots of people who are willing to help.

3. 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.

4. What are the data types in Node.js?

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

Primitives

- String
- Number
- Bigint
- Boolean
- Undefined
- Null
- Symbol

Objects

- Function
- Array
- Buffer: 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. Buffer is a class.
- other regular objects

5. 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

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Step 05: Run the app

```
node app.js
```

6. 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:

```
var url = require('url');
```

Then parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties.

```
var url = require('url');
var adr = 'http://localhost:8080/default.htm?year=2021&month=september';
var q = url.parse(adr, true);

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

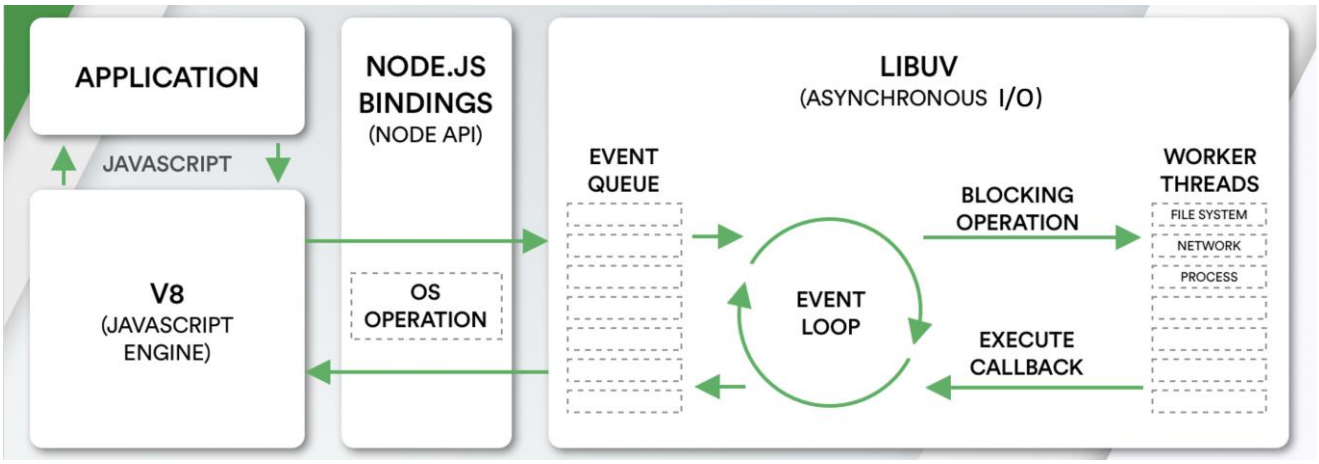
```
console.log(q.pathname); //returns '/default.htm'
console.log(q.search); //returns '?year=2021&month=september'

var qdata = q.query; //returns an object: { year: 2021, month: 'september' }
console.log(qdata.month); //returns 'september'
```

7. 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()
```

8. How do Node.js works?



Node 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.

So the server never needs to create additional threads or switch between threads, which means it has very little overhead. If you want to make full use of multiple hardware cores, you just start multiple instances of node.js

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.

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.

9. What is the difference between Node.js, AJAX, and JQuery?

Node.js is a javascript runtime that makes it possible for us to write back-end of applications.

Asynchronous JavaScript and XML(AJAX) refers to group of technologies that we use to send requests to web servers and retrieve data from them without reloading the page.

Jquery is a simple javascript library that helps us with front-end development.

10. What are the core modules of Node.js?

They are defined within the Node.js source and are located in the lib/ folder, and Node.js has several modules compiled into the binary.

Core modules are always preferentially loaded if their identifier is passed to require(). For instance, require('http') will always return the built in HTTP module, even if there is a file by that name.
Core modules can also be identified using the node: prefix, in which case it bypasses the require cache. For instance, require('node:http') will always return the built in HTTP module, even if there is require.cache entry by that name.

11. What is callback function in Node.js?

In node.js, we basically use callbacks for handling asynchronous operations like — making any I/O request, database operations or calling an API to fetch some data. Callback allows our code to not get blocked when a process is taking a long time.

```
function myNew(next){
   console.log("Im the one who initates callback");
   next("nope", "success");
}

myNew(function(err, res){
   console.log("I got back from callback",err, res);
});
```

12. What is callback hell in Node.js?

Callback hell is a phenomenon that afflicts a JavaScript developer when he tries to execute multiple asynchronous operations one after the other.
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.

```
getData(function(a){
   getMoreData(a, function(b){
      getMoreData(b, function(c){
         getMoreData(c, function(d){
```

```
                    getMoreData(d, function(e){
                            ...
                        });
                });
        });
    });
});
```

Techniques for avoiding callback hell

1. Using Async.js
2. Using Promises
3. Using Async-Await

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.

```
var outputPromise = getInputPromise().then(function (input) {
    //handle success
}, function (error) {
    //handle error
});
```

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.

```
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();
```

13. 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

```
var myPromise = new Promise(function(resolve, reject){
  ....
})
```

14. When should you npm and when yarn?

- **npm**

It is the default method for managing packages in the Node.js runtime environment. It relies upon a command line client and a database made up of public and premium packages known as the the npm registry. Users can access the registry via the client and browse the many packages available through the npm website. Both npm and its registry are managed by npm, Inc.

node -v
npm -v

- **Yarn**

Yarn was developed by Facebook in attempt to resolve some of npm's shortcomings. Yarn isn't technically a replacement for npm since it relies on modules from the npm registry. Think of Yarn as a new installer that still relies upon the same npm structure. The registry itself hasn't changed, but the installation method is different. Since Yarn gives you access to the same packages as npm, moving from npm to Yarn doesn't require you to make any changes to your workflow.

npm install yarn --global

15. 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:

var fs = require('fs');

```
var readFileStub = sinon.stub(fs, 'readFile', function (path, cb) {
  return cb(null, 'filecontent');
});

expect(readFileStub).to.be.called;
readFileStub.restore();
```

16. 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

17. 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

18. What is Event loop in Node.js? How does it work?

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.

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.

Event-Driven Programming

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.

Although events look quite similar to callbacks, the difference lies in the fact that 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 as follows

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

Example:

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

```
// Create an event handler as follows
var connectHandler = function connected() {
  console.log('connection succesful.');

  // Fire the data_received event
  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.");
```

19. What is REPL? What purpose it is used for?

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
>
```

20. 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.

21. 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.

- **Asynchronous VS Non-Blocking**

1. Asynchronous does not respond immediately, While Nonblocking responds immediately if the data is available and if not that simply returns an error.

2. Asynchronous improves the efficiency by doing the task fast as the response might come later, meanwhile, can do complete other tasks. Nonblocking does not block any execution and if the data is available it retrieves the information quickly.
3. Asynchronous is the opposite of synchronous while nonblocking I/O is the opposite of blocking. They both are fairly similar but they are also different as asynchronous is used with a broader range of operations while nonblocking is mostly used with I/O.

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

- **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.
- **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.
- **Jquery**: jQuery is a fast, small, and feature-rich JavaScript library.
- **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.
- **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.
- **Request**: Request is Simplified HTTP request client make it possible to make http calls. It supports HTTPS and follows redirects by default.

23. 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.

24. 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
});
```

25. 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.

26. What is typically the first argument passed to a Node.js callback handler?

The first argument to any callback handler is an optional error object

```
function callback(err, results) {
   // usually we'll check for the error before handling results
   if(err) {
      // handle error somehow and return
   }
   // no error, perform standard callback handling
}
```

27. 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

var http = require('http');
var 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
```

28. 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.

29. What is chrome v8 engine?

V8 is the name of the JavaScript engine that powers Google Chrome. It's the thing that takes our JavaScript and executes it while browsing with Chrome. V8 provides the runtime environment in which JavaScript executes. The DOM, and the other Web Platform APIs are provided by the browser.

V8 is Google's open source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and in Node.js, among others. It implements ECMAScript and WebAssembly, and runs on Windows 7 or later, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors. V8 can run standalone, or can be embedded into any C++ application.

30. Why is LIBUV needed in Node JS?

LIBUV is a library written in C and it's focus is on asynchronous I/O. Node.js use this library to interact with OS, system files and networking and also two core features of Node.js called event loop and thread pool are available in this runtime thanks to this library.

31. 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.

32. Why to use Express.js?

ExpressJS is a prebuilt NodeJS framework that can help you in creating server-side web applications faster and smarter. Simplicity, minimalism, flexibility, scalability are some of its characteristics and since it is made in NodeJS itself, it inherited its performance as well.

Express 3.x is a light-weight web application framework to help organize your web application into an MVC architecture on the server side. You can then use a database like MongoDB with Mongoose (for modeling) to provide a backend for your Node.js application. Express.js basically helps you manage everything, from routes, to handling requests and views.
It has become the standard server framework for node.js. Express is the backend part of something known as the MEAN stack. The MEAN is a free and open-source JavaScript software stack for building dynamic web sites and web applications which has the following components;

1. **MongoDB** - The standard NoSQL database
2. **Express.js** - The default web applications framework
3. **Angular.js** - The JavaScript MVC framework used for web applications
4. **Node.js** - Framework used for scalable server-side and networking applications.

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.

using Express

```
var express=require('express');
var app=express();
app.get('/',function(req,res) {
  res.send('Hello World!');
});
var server=app.listen(3000,function() {});
```

33. Write the steps for setting up an Express JS application?

1. Install Express Generator

C:\node>npm install -g express-generator

2. Create an Express Project

C:\node>express --view="ejs" nodetest1

3. Edit Dependencies

MAKE SURE TO CD INTO YOUR nodetest FOLDER. OK, now we have some basic structure in there, but we're not quite done. You'll note that the express-generator routine created a file called package.json in your nodetest1 directory. Open this up in a text editor and it'll look like this:

```
// C:\node\nodetest1\package.json
{
  "name": "nodetest1",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.9",
    "ejs": "~2.5.7",
    "express": "~4.16.0",
    "http-errors": "~1.6.2",
    "morgan": "~1.9.0"
  }
}
```

This is a basic JSON file describing our app and its dependencies. We need to add a few things to it. Specifically, calls for MongoDB and Monk.

C:\node\nodetest1>npm install --save monk@^6.0.6 mongodb@^3.1.13

4. Install Dependencies

C:\node\nodetest1>npm install

```
C:\node\nodetest1>npm start
Node Console

> nodetest1@0.0.0 start C:\node\nodetest1
> node ./bin/www
```

34. 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.

35. What are 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.

36. What is chaining process in Node.js?

It is an approach to connect the output of one stream to the input of another stream, thus creating a chain of multiple stream operations.

37. What is a control flow function? What are the steps does it execute?

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

It executes the following steps.

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

38. 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

39. When to use Node.js and when not to use it?

When to use Node.js

It is ideal to use Node.js for developing streaming or event-based real-time applications that require less CPU usage such as.

- Chat applications.
- Game servers -- Node.js is good for fast and high-performance servers, that face the need to handle thousands of user requests simultaneously.
- Good for A Collaborative Environment -- It is suitable for environments where multiple people work together. For example, they post their documents, modify them by doing check-out and check-in of these documents. Node.js supports such situations by creating an event loop for every change made to the document. The “Event loop” feature of Node.js enables it to handle multiple events simultaneously without getting blocked.
- Advertisement Servers -- Here again, we have servers that handle thousands of request for downloading advertisements from a central host. And Node.js is an ideal solution to handle such tasks.
- Streaming Servers -- Another ideal scenario to use Node.js is for multimedia streaming servers where clients fire request’s towards the server to download different multimedia contents from it.

To summarize, it's good to use Node.js, when you need high levels of concurrency but less amount of dedicated CPU time.

Last but not the least, since Node.js uses JavaScript internally, so it fits best for building client-side applications that also use JavaScript.

When to not use Node.js

However, 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.

40. How to make post request in Node.js?

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)
      }
   });
```

41. Can you create http server in Node.js, explain the code used for it?

Yes, we can create HTTP Server in Node.js. We can use the <http-server> command to do so.
Following is the sample code.

```
var http = require('http');
var requestListener = function (request, response) {
   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Welcome Viewers\n');
}
var server = http.createServer(requestListener);
server.listen(4200); // The port where you want to start with.
```

42. How to load html in Node.js?

To load HTML in Node.js we have to change the “Content-type” in the HTML code from text/plain to text/html.

```
fs.readFile(filename, "binary", function(err, file) {
   if(err) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(err + "\n");
      response.end();
      return;
   }

response.writeHead(200);
response.write(file, "binary");
response.end();
});
```

Now we will modify this code to load an HTML page instead of plain text.

```
fs.readFile(filename, "binary", function(err, file) {
   if(err) {
      response.writeHead(500, {"Content-Type": "text/html"});
      response.write(err + "\n");
      response.end();
      return;
   }

response.writeHead(200, {"Content-Type": "text/html"});
response.write(file);
response.end();
});
```

43. What is your favourite HTTP framework and why?

Express.js

Express provides a thin layer on top of Node.js with web application features such as basic routing, middleware, template engine and static files serving, so the drastic I/O performance of Node.js doesn't get compromised.

Express is a minimal, un-opinionated framework. it doesn't apply any of the prevalent design patterns such as MVC, MVP, MVVM or whatever is trending out of the box. For fans of simplicity, this is a big plus among all other frameworks because you can build your application with your own preference and no unnecessary learning curve. This is especially advantageous when creating a new personal project with no historical burden, but as the project or developing team grows, lack of standardization may lead to extra work for project/code management, and worst case scenario it may lead to the inability to maintain.

Generator

Even though the framework is un-opinionated, it does have the generator that generates specific project folder structure. After installing express-generator npm package and creating application skeleton with generator command, an application folder with clear hierarchy will be created to help you organize images, front-end static JavaScript, stylesheet files and HTML template files.

```
npm install express-generator -g
express helloapp
```

Middleware

Middleware are basically just functions that have full access to both request and response objects.

```
var app = express();


app.use(cookieParser());
app.use(bodyParser());
app.use(logger());
app.use(authentication());


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

```
app.listen(3000);
```

An Express application is essentially Node.js with a host of middleware functions, whether you want to customize your own middleware or take advantage of the built-in middlewares of the framework, Express made the process natural and intuitive.

Template Engine

Template engines allow developer to embed backend variables into HTML files, and when requested the template file will be rendered to plain HTML format with the variables interpolated with their actual values. By default, the express-generator uses Pug (originally known as Jade) template engine, but other options like Mustache and EJS also work with Express seamlessly.

Database Integration

As a minimal framework, Express does not consider database integration as a required aspect within its package, thus it leans toward no specific database usage whatsoever. While adopting a particular data storage technology, be it MySQL, MongoDB, PostgreSQL, Redis, ElasticSearch or something else, it's just a matter of installing the particular npm package as database driver. These third party database drivers do not conform to unified syntax when doing CRUD instructions, which makes switching databases a big hassle and error prone.

44. What are the Challenges with Node.js?

Challenges with Node.js Application Maintenance

Improper maintenance of an application can result in issues related to stability or flexibility, often leading to the app's failure. If the code is not well-written or if developers use outdated tools, the performance can suffer, and users might experience more bugs and app crashes. On top of that, poor-quality code can hamper the app's scaling capacity and the further development of the application. In the worst case scenario, it might become impossible to introduce new features without rewriting the codebase from scratch.

1. Extensive stack
2. Technical Debt
3. Scalability challanges
4. Poor documentation

45. What is the difference between Node.js vs Ajax?

1. AJAX

AJAX stands for Asynchronous Javascript and XML, it's used to allow web pages (client-side) to update asynchronously by communicating with a web server and by exchanging data. This essentially means that applications can talk to a server in the background of the application. It uses some core components to function:

1. The browser XMLHttpRequest object to request data from a server
2. HTML/CSS to display or collect data
3. DOM for dynamic display
4. JSON/XML for interchanging the data
5. Javascript to unify everything

2. Node.js

Node.js allows the developers to develop a web application in a single language called JavaScript for both client side and server side.

Unlike the other programming languages, Node.js has its cycle of the event in the form of language which is very beneficial for high-performance and scalable application development.

It is required for those web applications where traffic rate is very high. Node.js is an event based I/O language and its response time is very high rather than the other traditional languages. It is being used by the famous websites like Linked in, Twitter and Gmail.

The runtime environment of Node.js interprets JavaScript, which is very easy and simple to understand and code. Due to this reason, even the developers find it easy going which keeps them happy and relaxed. It is pertinent for real-time collaborative apps.

46. How Node.js overcomes the problem of blocking of I/O operations?

Node.js solves this problem by putting the event based model at its core, using an event loop instead of threads.

Node.js uses an event loop for this. An event loop is “an entity that handles and processes external events and converts them into callback invocations”. Whenever data is needed nodejs registers a callback and sends the operation to this event loop. Whenever the data is available the callback is called.

47. 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.

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;
}
```

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);
});
```

48. What is LTS releases of Node.js why should you care?

An LTS(Long Term Support) version of Node.js receives all the critical bug fixes, security updates and performance

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.

49. 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. 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);
```

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

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.

51. What is difference between JavaScript and Node.js?

	JavaScript	**Node JS**
Type	JavaScript is a programming language. It running in any web browser with a proper browser engine.	It is an interpreter and environment for JavaScript with some specific useful libraries which JavaScript programming can use separately.

	JavaScript	Node JS
Utility	Mainly using for any client-side activity for a web application, like possible attribute validation or refreshing the page in a specific interval or provide some dynamic changes in web pages without refreshing the page.	It mainly used for accessing or performing any non-blocking operation of any operating system, like creating or executing a shell script or accessing any hardware specific information or running any backend job.
Running Engine	JavaScript running any engine like Spider monkey (FireFox), JavaScript Core (Safari), V8 (Google Chrome).	Node JS only run in a V8 engine which mainly used by google chrome. And javascript program which will be written under this Node JS will be always run in V8 Engine.

52. What are the difference between Events and Callbacks?

Node.js is a single-threaded application, but it 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 thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

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

1. Callback: 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: synchronous callback

```
function greeting(name) {
  alert('Hello ' + name);
}

function processUserInput(callback) {
  var name = prompt('Please enter your name.');
  callback(name);
}
```

processUserInput(greeting);

2. Events: Every action on a computer is an event. Node.js allows us to create and handle custom events easily by using events module. Event module includes EventEmitter class which can be used to raise and handle custom events.

Example:

```
var event = require('events');
var eventEmitter = new event.EventEmitter();

//  Add listener function for Sum event
eventEmitter.on('Sum', function(num1, num2) {
   console.log('Total: ' + (Number(num1) + Number(num2)));
});

//  Call Event.
eventEmitter.emit('Sum', '10', '20');
```

53. 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

Principles of REST

- Uniform Interface
- Stateless

- Cacheable
- Client-Server
- Layered System
- Code on Demand (optional)

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 −

```
var express = require('express');
var app = express();
var fs = require("fs");

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

```
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
     console.log( data );
     res.end( data );
  });
})

var server = app.listen(3000, function () {
  var host = server.address().address
  var 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.

```
var express = require('express');
var app = express();
var fs = require("fs");

var 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));
  });
})
```

```
var server = app.listen(3000, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("App listening at http://%s:%s", host, port)
})
```

Delete User

```
var express = require('express');
var app = express();
var fs = require("fs");

var 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));
  });
})

var server = app.listen(3000, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("App listening at http://%s:%s", host, port)
})
```

54. What is the difference between req.params and req.query?

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 query string 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

55. How to handle file upload in Node.js?

- **express**: Popular web framework built on top of Node.js, used for creating REST-API.
- **multer**: Middleware for handling multipart/form-data — file uploads

1. Installing the dependencies

npm install express multer --save

2. Package.json

```
{
  "name": "file_upload",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.13.3",
    "multer": "1.1.0"
  },
  "devDependencies": {
    "should": "~7.1.0",
    "mocha": "~2.3.3",
    "supertest": "~1.1.0"
  }
}
```

3. Server.js

```
var express = require("express");
var multer  = require('multer');
var app = express();

// for text/number data transfer between clientg and server
app.use(express());

var storage =   multer.diskStorage({
```

```
  destination: function (req, file, callback) {
    callback(null, './uploads');
  },
  filename: function (req, file, callback) {
    callback(null, file.fieldname + '-' + Date.now());
  }
});

var 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");
});
```

4. 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>
```

56. Explain the terms nodemon, cors, dotenv, moment in Express JS?

a) 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"
  },
  // ...
}
```

b) cors

Cross-**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. CORS defines a way in

which a browser and server can interact and determine whether or not it is safe to allow a cross-origin request.

Installation

npm install cors

Example: Enable All CORS Requests

```
var express = require('express')
var cors = require('cors')
var app = express()

app.use(cors())

app.get('/products/:id', function (req, res, next) {
  res.json({msg: 'This is CORS-enabled for all origins!'})
})

app.listen(8080, function () {
  console.log('CORS-enabled web server listening on port 80')
})
```

Example: Enable CORS for a Single Route

```
var express = require('express')
var cors = require('cors')
var 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')
})
```

c) 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
})
```

d) 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 2020, 3:15:22 pm
moment().format('dddd');                    // Saturday
```

```
moment().format("MMM Do YY");              // Oct 24th 20
```

- 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(); // 10/14/2020
moment().subtract(6, 'days').calendar();  // Last Sunday at 3:18 PM
moment().subtract(3, 'days').calendar();  // Last Wednesday at 3:18 PM
```

57. 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

```
var express = require('express')
var 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
```

58. 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);
  }
}
```

59. 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.

JSON Web Tokens consist of three parts separated by dots (.), which are:

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

```
var express = require('express');
var router = express.Router();
var bodyParser = require('body-parser');
var User = require('../user/User');

var jwt = require('jsonwebtoken');
var bcrypt = require('bcryptjs');
var config = require('../config');



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

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

  var 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
    var 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.

60. 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.

61. What are the middleware functions in Node.js?

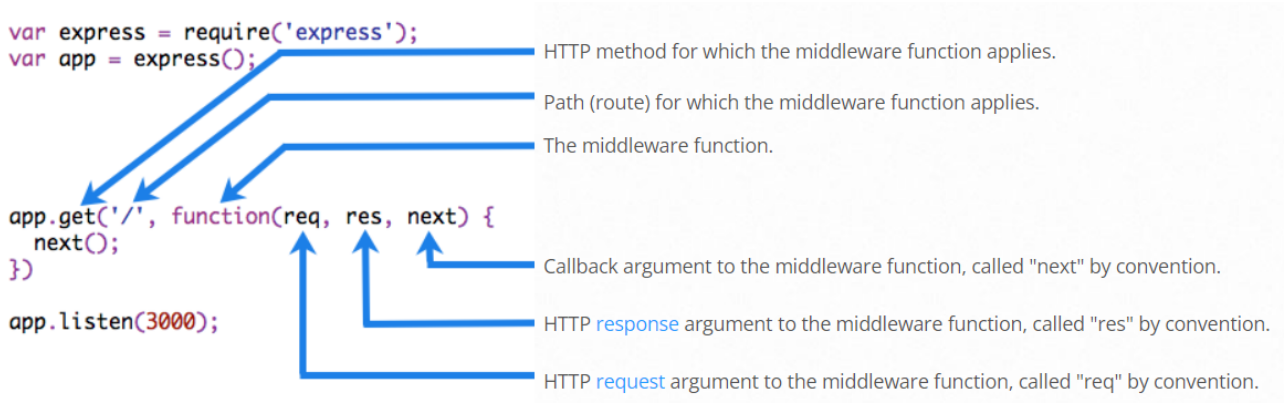
Middleware functions are functions that have access to the **request object (req)**, the **response object (res)**, and the next function in the application's request-response cycle.
Middleware functions can perform the following tasks:

- 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:



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.