Q-1. What is Node.js?

Answer.

Node.js is a JavaScript runtime or platform which is built on Google Chrome's JavaScript v8 engine. This runtime allows executing the JavaScript code on any machine outside a browser (this means that it is the server that executes the Javascript and not the browser).

Node.js is single-threaded, that employs a concurrency model based on an event loop. It doesn't block the execution instead registers a callback which allows the application to continue. It means Node.js can handle concurrent operations without creating multiple threads of execution so can scale pretty well.

It uses JavaScript along with C/C++ for things like interacting with the filesystem, starting up HTTP or TCP servers and so on. Due to it's extensively fast growing community and NPM, Node.js has become a very popular, open source and cross-platform app. It allows developing very fast and scalable network app that can run on Microsoft Windows, Linux, or OS X.

Following are the areas where it's perfect to use Node.js.

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

At the same time, it's not suitable for heavy applications involving more of CPU usage.

Q-2. What are the key features of Node.js?

Answer.

Let's look at some of the key features of Node.js.

- **Asynchronous event driven IO helps concurrent request handling –** All APIs of Node.js are asynchronous. This feature means that if a Node receives a request for some Input/Output operation, it will execute that operation in the background and continue with the processing of other requests. Thus it will not wait for the response from the previous requests.
- **Fast in Code execution –** Node.js uses the V8 JavaScript Runtime engine, the one which is used by Google Chrome. Node has a wrapper over the JavaScript engine which makes the runtime engine much faster and hence processing of requests within Node.js also become faster.
- **Single Threaded but Highly Scalable –** Node.js uses a single thread model for event looping. The response from these events may or may not reach the server immediately. However, this does not block other operations. Thus making Node.js highly scalable. Traditional servers create limited threads to handle requests while

Node.js creates a single thread that provides service to much larger numbers of such requests.

- **Node.js library uses JavaScript –** This is another important aspect of Node.js from the developer's point of view. The majority of developers are already well-versed in JavaScript. Hence, development in Node.js becomes easier for a developer who knows JavaScript.
- **There is an Active and vibrant community for the Node.js framework –** The active community always keeps the framework updated with the latest trends in the web development.
- **No Buffering –** Node.js applications never buffer any data. They simply output the data in chunks.

Q-3. Explain how do we decide, when to use Node.js and when not to use it?

Answer.

When should we use Node.js?

It's 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.

Q-4. What IDEs can you use for Node.js development?

Answer.

Here is the list of most commonly used IDEs for developing node.js applications.

Cloud9.

It is a free, cloud-based IDE that supports, application development, using popular programming languages like Node.js, PHP, C++, Meteor and more. It provides a powerful online code editor that enables a developer to write, run and debug the app code.

JetBrains WebStorm.

WebStorm is a lightweight yet powerful JavaScript IDE, perfectly equipped for doing client-side and server-side development using Node.js. The IDE provides features like intelligent code completion, navigation, automated and safe refactorings. Additionally, we can use the debugger, VCS, terminal and other tools present in the IDE.

JetBrains InteliJ IDEA.

It is a robust IDE that supports web application development using mainstream technologies like Node.js, Angular.js, JavaScript, HTML5 and more. To enable the IDE that can do Node.js development we have to install a Node.js plugin. It provides features, including syntax highlighting, code assistance, code completion and more. We can even run and debug Node.js apps and see the results right in the IDE. It's JavaScript debugger offers conditional breakpoints, expression evaluation, and other features.

Komodo IDE.

It is a cross-platform IDE that supports development in main programming languages, like Node.js, Ruby, PHP, JavaScript and more. It offers a variety of features, including syntax highlighting, keyboard shortcuts, collapsible Pane, workspace, auto indenting, code folding and code preview using built-in browser.

Eclipse.

It is a popular cloud-based IDE for web development using Java, PHP, C++ and more. You can easily avail the features of Eclipse IDE using the Node.js plug-in, which is <**nodeclipse**>.

Atom.

It is an open source application built with the integration of HTML, JavaScript, CSS, and Node.js. It works on top of Electron framework to develop cross-platform apps using web technologies. Atom comes pre-installed with four UI and eight syntax themes in both dark and light colors. We can also install themes created by the Atom community or create our own if required.

Q-5. Explain how does Node.js work?

Answer.

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

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

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

Let's take an example of a grocery delivery.

Usually, the delivery boy goes to each and every house to deliver the packet. Node.js works in the same way and processes one request at a time. The problem arises when any one house is not open. The delivery boy can't stop at one house and wait till it gets opened up. What he will do next, is to call the owner and ask him to call when the house is open. Meanwhile, he is going to other places for delivery. Node.js works in the same way. It doesn't wait for the processing of the request to complete (house is open). Instead, it attaches a callback function (call from the owner of the house) to it. Whenever the processing of a request completes (the house is open), an event gets called, which triggers the associated callback function to send the response.

To summarize, Node.js does not process the requests in parallel. Instead, all the back-end processes like, I/O operations, heavy computation tasks, that take a lot of time to execute, run in parallel with other requests.

Q-6. Explain REPL in Node.js?

Answer.

The REPL stands for “Read Eval Print Loop”. It is a simple program that accepts the commands, evaluates them, and finally prints the results. REPL provides an environment similar to that of Unix/Linux shell or a window console, in which we can enter the command and the system, in turn, responds with the output. REPL performs the following tasks.

- **READ**
  - It Reads the input from the user, parses it into JavaScript data structure and then stores it in the memory.
- **EVAL**
  - It Executes the data structure.
- **PRINT**
  - It Prints the result obtained after evaluating the command.
- **LOOP**
  - It Loops the above command until the user presses Ctrl+C two times.

Q-7. Is Node.js entirely based on a single-thread?

Answer.

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

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

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

Q-8. How to get Post Data in Node.js?

Answer.

Following is the code snippet to fetch Post Data using Node.js.

```
app.use(express.bodyParser());
app.post('/', function(request, response){
console.log(request.body.user);
});
```

Q-9. How to make Post request in Node.js?

Answer.

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

```
var request = require('request');
request.post(
'http://www.example.com/action',
{ form: { key: 'value' } },
function (error, response, body) {
if (!error && response.statusCode == 200) {
console.log(body)
}
}
);
```

Q-10. What is Callback in Node.js?

Answer.

We may call “callback” as an asynchronous equivalent for a function. Node.js makes heavy use of callbacks and triggers it at the completion of a given task. All the APIs of Node.js are written in such a way that they support callbacks.

For example, suppose we have a function to read a file, as soon as it starts reading the file, Node.js return the control immediately to the execution environment so that the next instruction can be executed. Once file read operation is complete, it will call the callback function and pass the contents of the file as its arguments. Hence, there is no blocking or wait, due to File I/O. This functionality makes Node.js as highly scalable, using it processes a high number of requests without waiting for any function to return the expected result.

Q-11. What is Callback Hell?

Answer.

Initially, you may praise Callback after learning about it. Callback hell is heavily nested callbacks which make the code unreadable and difficult to maintain.

Let’s see the following code example.

```
downloadPhoto('http://coolcats.com/cat.gif', displayPhoto)
function displayPhoto (error, photo) {
if (error) console.error('Download error!', error)
else console.log('Download finished', photo)
}
```

```
console.log('Download started')
```

In this scenario, Node.js first declares the “displayPhoto” function. After that, it calls the “downloadPhoto” function and pass the “displayPhoto” function as its callback. Finally, the code prints ‘Download started’ on the console. The “displayPhoto” will be executed only after “downloadPhoto” completes the execution of all its tasks.

Q-12. How to avoid callback hell in Node.js?

Answer.

Node.js internally uses a single-threaded event loop to process queued events. But this approach may lead to blocking the entire process if there is a task running longer than expected.

Node.js addresses this problem by incorporating callbacks also known as higher-order functions. So whenever a long-running process finishes its execution, it triggers the callback associated. With this approach, it can allow the code execution to continue past the long-running task.

However, the above solution looks extremely promising. But sometimes, it could lead to complex and unreadable code. More the no. of callbacks, longer the chain of returning callbacks would be. Just see the below example.

With such an unprecedented complexity, it’s hard to debug the code and can cause you a whole lot of time. There are four solutions which can address the callback hell problem.

1. Make your program modular.

It proposes to split the logic into smaller modules. And then join them together from the main module to achieve the desired result.

2. Use async mechanism.

It is a widely used Node.js module which provides a sequential flow of execution.

The async module has <async.waterfall> API which passes data from one operation to other using the next callback.

Another async API <async.map> allows iterating over a list of items in parallel and calls back with another list of results.

With the async approach, the caller’s callback gets called only once. The caller here is the main method using the async module.

3. Use promises mechanism.

Promises give an alternate way to write async code. They either return the result of execution or the error/exception. Implementing promises requires the use of <.then()> function which waits for the promise object to return. It takes two optional arguments, both functions. Depending on the state of the promise only one of them will get called. The first function call proceeds if the promise gets fulfilled. However, if the promise gets rejected, then the second function will get called.

4. Use generators.

Generators are lightweight routines, they make a function wait and resume via the yield keyword. Generator functions uses a special syntax <function* ()>. They can also suspend and resume asynchronous operations using constructs such as promises or <thunks> and turn a synchronous code into asynchronous.

Q-13. Can you create HTTP Server in Nodejs, explain the code used for it?

Answer.

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(8080); // The port where you want to start with.
```

Q-14. What is the difference between Nodejs, AJAX, and jQuery?

Answer.

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

Node.js –

It is a server-side platform for developing client-server applications. For example, if we've to build an online employee management system, then we won't do it using client-side JS. But the Node.js can certainly do it as it runs on a server similar to Apache, Django not in a browser.

AJAX (aka Asynchronous Javascript and XML) –

It is a client-side scripting technique, primarily designed for rendering the contents of a page without refreshing it. There are a no. of large companies utilizing AJAX such as Facebook and Stack Overflow to display dynamic content.

jQuery –

It is a famous JavaScript module which complements AJAX, DOM traversal, looping and so on. This library provides many useful functions to help in JavaScript development. However, it's not mandatory to use it but as it also manages cross-browser compatibility, so can help you produce highly maintainable web applications.

Q-15. What are Globals in Node.js?

Answer.

There are three keywords in Node.js which constitute as Globals. These are Global, Process, and Buffer.

Global.

The Global keyword represents the global namespace object. It acts as a container for all other <global> objects. If we type <console.log(global)>, it'll print out all of them.

An important point to note about the global objects is that not all of them are in the global scope, some of them fall in the module scope. So, it's wise to declare them without using the var keyword or add them to Global object.

Variables declared using the var keyword become local to the module whereas those declared without it get subscribed to the global object.

Process.

It is also one of the global objects but includes additional functionality to turn a synchronous function into an async callback. There is no boundation to access it from anywhere in the code. It is the instance of the EventEmitter class. And each node application object is an instance of the Process object.

It primarily gives back the information about the application or the environment.

- **<process.execPath>** – to get the execution path of the Node app.
- **<process.Version> –** to get the Node version currently running.
- **<process.platform> –** to get the server platform.

Some of the other useful Process methods are as follows.

- **<process.memoryUsage> –** To know the memory used by Node application.
- **<process.NextTick> –** To attach a callback function that will get called during the next loop. It can cause a delay in executing a function.

Buffer.

The Buffer is a class in Node.js to handle binary data. It is similar to a list of integers but stores as a raw memory outside the V8 heap.

We can convert JavaScript string objects into Buffers. But it requires mentioning the encoding type explicitly.

- **<ascii> –** Specifies 7-bit ASCII data.
- **<utf8> –** Represents multibyte encoded Unicode char set.
- **<utf16le> –** Indicates 2 or 4 bytes, little endian encoded Unicode chars.
- **<base64> –** Used for Base64 string encoding.
- **<hex> –** Encodes each byte as two hexadecimal chars.

Here is the syntax to use the Buffer class.

```
> var buffer = new Buffer(string, [encoding]);
```

The above command will allocate a new buffer holding the string with <utf8> as the default encoding. However, if you like to write a <string> to an existing buffer object, then use the following line of code.

```
> buffer.write(string)
```

This class also offers other methods like <readInt8> and <writeUInt8> that allows read/write from various types of data to the buffer.

Q-16. How to load HTML in Node.js?

Answer.

To load HTML in Node.js we have to change the “Content-type” in the HTML code from text/plain to text/html.
Let’s see an example where we have created a static file in web server.

```
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();
});
```

Q-17. What is EventEmitter in Node.js?

Answer.

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

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

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

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

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

Q-18. How many types of Streams are present in Node.js?

Answer.

Stream in Node.js are objects that allow reading data from a source or writing data to a specific destination in a continuous fashion. In Node.js, there are four types of streams.

- **<Readable> –** This is the Stream to be used for reading operation.
- **<Writable> –** It facilitates the write operation.
- **<Duplex> –** This Stream can be used for both the read and write operations.
- **<Transform> –** It is a form of a duplex Stream, which performs the computations based on the available input.

All the Streams, discussed above are an instance of an “EventEmitter” class. The event thrown by the Stream varies with time. Some of the commonly used events are as follows.

- **<data> –** This event gets fired when there is data available for reading.
- **<end> –** The Stream fires this event when there is no more data to read.
- **<error> –** This event gets fired when there is any error in reading or writing data.
- **<finish> –** It fires this event after it has flushed all the data to the underlying system.

Q-19. List and Explain the important REPL commands?

Answer.

Following is the list of some of the most commonly used REPL commands.

- **<.help> –** It displays help for all the commands.
- **<tab Keys> –** It displays the list of all the available commands.
- **<Up/Down Keys> –** Its use is to determine what command was executed in REPL previously.
- **<.save filename> –** Save the current REPL session to a file.
- **<.load filename> –** To Load the specified file in the current REPL session.
- **<ctrl + c> –** used to Terminate the current command.
- **<ctrl + c (twice)> –** To Exit from the REPL.
- **<ctrl + d> –** This command perfoms Exit from the REPL.
- **<.break> –** It leads Exitting from multiline expression.
- **<.clear> –** Exit from multiline expression.

Q-20. What is NPM in Node.js?

Answer.

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
```

Q-21. What is the global installation of dependencies?

Answer.

Globally installed packages/dependencies are stored in <user-directory>/npm directory. Such dependencies can be used in CLI (Command Line Interface) function of any node.js, but cannot be imported using require() in the Node application directly.

To install a Node project globally use -g flag as.

```
C:\Nodejs_WorkSpace>npm install express -g
```

Q-22. What is the local installation of dependencies?

Answer.

By default, NPM installs any dependency in the local mode. It means that the package gets installed in "node_modules" directory which is present in the same folder, where Node application is placed. Locally deployed packages are accessible via require(). Following is the syntax to install a Node project locally.

```
C:\Nodejs_WorkSpace>npm install express
```

Q-23. What is package.json? Who uses it?

Answer.

What is <package.json>?

- It is a plain JSON (JavaScript Object Notation) text file which contains all metadata information about Node.js Project or application.
- This file should be present in the root directory of every Node.js Package or Module to describe its metadata in JSON format.
- The file is named as "package" because Node.js platform treats every feature as a separate component. Node.js calls these as Package or Module.

Who use it?

- NPM (Node Package Manager) uses <package.json> file. It includes details of the Node.js application or package. This file contains a no. of different directives or elements. These directives guide NPM, about how to handle a module or package.

Q-24. Does Node.js support multi-core platforms? And is it capable of utilizing all the cores?

Answer.

Yes, Node.js would run on a multi-core system without any issue. But it is by default a single-threaded application, so it can't completely utilize the multi-core system.

However, Node.js can facilitate deployment on multi-core systems where it does use the additional hardware. It packages with a Cluster module which is capable of starting multiple Node.js worker processes that will share the same port.

Q-25. Which is the first argument usually passed to a Node.js callback handler?

Answer.

Node.js core modules follow a standard signature for its callback handlers and usually the first argument is an optional error object. And if there is no error, then the argument defaults to null or undefined.

Here is a sample signature for the Node.js callback handler.

```
function callback(error, results) {
// Check for errors before handling results.
if ( error ) {
// Handle error and return.
}
// No error, continue with callback handling.
}
```

Q-26. What is chaining process in Node.js?

Answer.

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

Q-27. How to create a custom directive in AngularJS?

Answer.

To create a custom directive, we have to first register it with the application object by calling the <directive> function. While invoking the <register> method of <directive>, we need to give the name of the function implementing the logic for that directive.

For example, in the below code, we have created a copyright directive which returns a copyright text.

```
app.directive('myCopyRight', function ()
{
return
{
template: '@CopyRight MyDomain.com '
};
});
```

Note – A custom directive should follow the camel case format as shown above.

Q-28. What is a child_process module in Node.js?

Answer.

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

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

Node.js provides a <child_process> module which supports following three methods to create a child process.

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

Q-29. What are the different custom directive types in AngularJS?

Answer.

AngularJS supports a no. of different directives which also depend on the level we want to restrict them.

So in all, there are four different kinds of custom directives.

- Element Directives (E)
- Attribute Directives (A)

- CSS Class Directives (C)
- Comment Directives (M)

Q-30. What is a control flow function? What are the steps does it execute?

Answer.

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.