Interview Questions

Node.js

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

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

Source: *tutorialspoint.com*

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

Answer: Following are main benefits of using Node.js

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

Source: *tutorialspoint.com*

Q3: Is Node a single threaded application? ☆☆

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

Source: *tutorialspoint.com*

Q4: What is global installation of dependencies? ☆☆

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

Source: *tutorialspoint.com*

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

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

```
fs.readFile(filePath, function(err, data) {
  if (err) {
    //handle the error
  }
  // use the data object
});
```

Source: *tutorialspoint.com*

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

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

Source: *blog.risingstack.com*

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

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

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

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

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

Source: *techbeamers.com*

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

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

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

Source: *techbeamers.com*

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

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

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

- **There is an Active and vibrant community for the Node.js framework –** The active community always keeps the framework updated with the latest trends in the web development.
- **No Buffering –** Node.js applications never buffer any data. They simply output the data in chunks.

Source: *techbeamers.com*

Q10: What is control flow function? ☆☆

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

Source: *lazyquestion.com*

Q11: What are Event Listeners? ☆☆

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

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

Source: *lazyquestion.com*

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

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

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

This is how Node manages concurrency.

Source: *codeforgeek.com*

Q13: What is Callback Hell? ☆☆

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

Source: *codeforgeek.com*

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

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

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

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

Source: *codeforgeek.com*

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

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

Source: *a4academics.com*

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

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

Source: *codingdefined.com*

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

Answer:

- EventEmitter
- Stream
- FS
- Net

- Global Objects

Source: *github.com/jimuyouyou*

Q18: What is V8? ☆☆

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

The Chrome V8 engine :

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

Source: *nodejs.org*

Q19: What is libuv? ☆☆

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

Source: *nodejs.org*

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

Answer:

```
return callback();
//some more lines of code; -  won't be executed

callback();
//some more lines of code; - will be executed
```

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

```
function do2(callback) {
    log.trace('Execute function: do2');
    return callback('do2 callback param');
}
```

```
var do2Result = do2((param) => {
    log.trace(`print ${param}`);
    return `return from callback(${param})`; // we could use that return
});

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

Output:

```
C:\Work\Node>node --use-strict main.js
[0] Execute function: do2
[0] print do2 callback param
[0] print return from callback(do2 callback param)
```

Source: *stackoverflow.com*

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

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

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

Source: *tutorialspoint.com*

Q22: What is Callback? ☆☆☆

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

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

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

Source: *tutorialspoint.com*

Q23: What is a blocking code? ☆☆☆

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

Source: *tutorialspoint.com*

Q24: How Node prevents blocking code? ☆☆☆

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

Source: *tutorialspoint.com*

Q25: What is Event Loop? ☆☆☆

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

Source: *tutorialspoint.com*

Q26: What is Event Emmitter? ☆☆☆

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

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

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

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

Source: *tutorialspoint.com*

Q27: What is purpose of Buffer class in Node? ☆☆☆

Answer: Buffer class is a global class and can be accessed in application without importing buffer module. A Buffer is a kind of an array of integers and corresponds to a raw memory allocation outside the V8 heap. A Buffer cannot be resized.

Source: *tutorialspoint.com*

Q28: What is difference between synchronous and asynchronous method of fs module? ☆☆☆

Answer:

Every method in `fs` module has synchronous as well as asynchronous form. Asynchronous methods takes a last parameter as completion function callback and first parameter of the callback function is error. It is preferred to use asynchronous method instead of synchronous method as former never block the program execution where the latter one does.

Source: *tutorialspoint.com*

Q29: What are streams? ☆☆☆

Answer: Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams.

- **Readable** - Stream which is used for read operation.
- **Writable** - Stream which is used for write operation.
- **Duplex** - Stream which can be used for both read and write operation.
- **Transform** - A type of duplex stream where the output is computed based on input.

Source: *tutorialspoint.com*

Q30: What is Chaining in Node? ☆☆☆

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

Source: *tutorialspoint.com*

Q31: What is the purpose of setTimeout function? ☆☆☆

Answer: The `setTimeout(cb, ms)` global function is used to run callback `cb` after at least `ms` milliseconds. The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days.

Source: *tutorialspoint.com*

Q32: How can you avoid callback hells? ☆☆☆

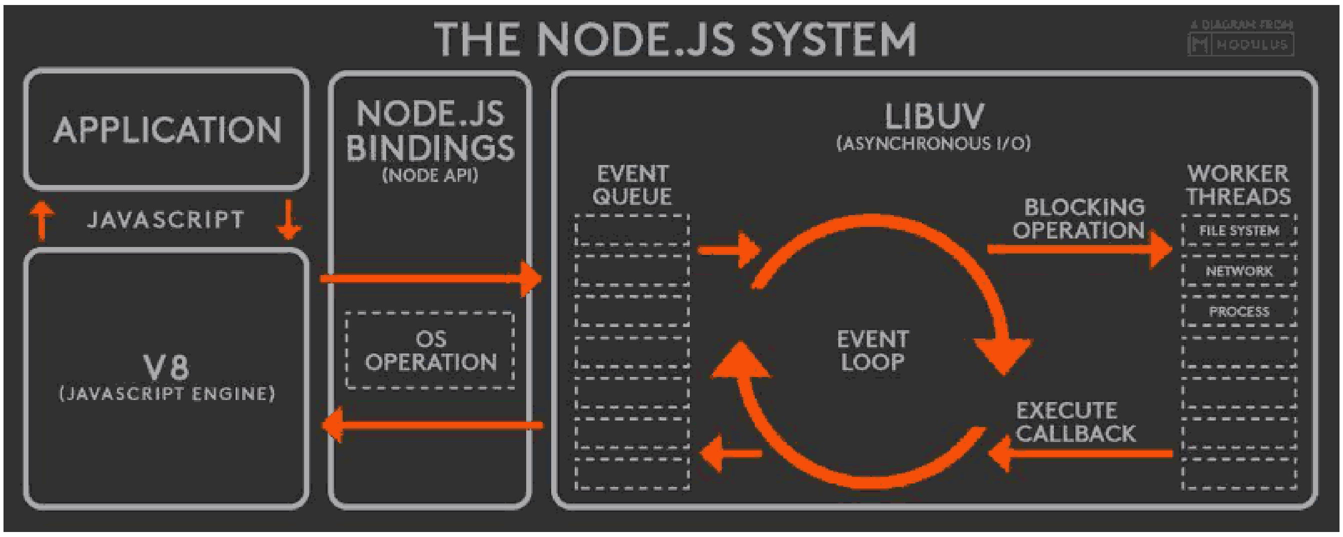
Answer: To do so you have more options:

- **modularization**: break callbacks into independent functions
- use *Promises*
- use `yield` with *Generators* and/or *Promises*

Source: *tutorialspoint.com*

Q33: What's the event loop? ☆☆☆

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



Every I/O requires a callback - once they are done they are pushed onto the event loop for execution. Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.

Source: *blog.risingstack.com*

Q34: How to avoid callback hell in Node.js? ☆☆☆

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

Node.js addresses this problem by incorporating callbacks also known as higher-order functions. So whenever a long-running process finishes its execution, it triggers the callback associated.

sometimes, it could lead to complex and unreadable code. More the no. of callbacks, longer the chain of returning callbacks would be.

There are four solutions which can address the callback hell problem.

Make your program modular

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

Use async mechanism

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

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

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

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

Use promises mechanism

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

Use generators

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

Source: *techbeamers.com*

Q35: Explain how does Node.js work? ☆☆☆

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

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

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

Source: *techbeamers.com*

Q16: When should we use Node.js? ☆☆☆

Answer: Node.js is well suited for applications that have a lot of concurrent connections and each *request only needs very few CPU cycles*, because the event loop (with all the other clients) is blocked during execution of a function. I believe Node.js is best suited for real-time applications: online games, collaboration tools, chat rooms, or anything where what one user (or robot? or sensor?) does with the application needs to be seen by other users immediately, without a page refresh.

Source: *techbeamers.com*

Q17: How does Node.js handle child threads? ☆☆☆

Answer: Node.js, in its essence, is a single thread process. It does not expose child threads and thread management methods to the developer. Technically, Node.js does spawn child threads for certain tasks such as asynchronous I/O, but these run behind the scenes and do not execute any application JavaScript code, nor block the main event loop.

If threading support is desired in a Node.js application, there are tools available to enable it, such as the ChildProcess module.

Source: *lazyquestion.com*

Q18: What is the preferred method of resolving unhandled exceptions in Node.js? ☆☆☆

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

```
process.on('uncaughtException', function(err) {
  console.log('Caught exception: ' + err);
});
```

However, `uncaughtException` is a very crude mechanism for exception handling and may be removed from Node.js in the future. An exception that has bubbled all the way up to the `Process` level means that your application, and Node.js may be in an undefined state, and the only sensible approach would be to restart everything.

The preferred way is to add another layer between your application and the Node.js process which is called the domain.

Domains provide a way to handle multiple different I/O operations as a single group. So, by having your application, or part of it, running in a separate domain, you can safely handle exceptions at the domain level, before they reach the `Process` level.

Source: *lazyquestion.com*

Q19: What is stream and what are types of streams available in Node.js? ☆☆☆

Answer: Streams are a collection of data that might not be available all at once and don't have to fit in memory. Streams provide chunks of data in a continuous manner. It is useful to read a large set of data and process it.

There is four fundamental type of streams:

- Readable.
- Writeable.
- Duplex.
- Transform.

Readable streams as the name suggest used in reading a large chunk of data from a source. Writable streams are used in writing a large chunk of data to the destination.

Duplex streams are both readable and writable (Eg socket). Transform stream is the duplex stream which is used in modifying the data (eg zip creation).

Source: *codeforgeek.com*

Q20: What are the global objects of Node.js? ☆☆☆

Answer: These objects are available in all modules:

- **process** - The process object is a global that provides information about, and control over, the current Node.js process.
- **console** - Used to print to stdout and stderr.
- **buffer** - Used to handle binary data.

Source: *github.com/jimuyouyou*

Q1: What is Piping in Node? ☆☆☆☆

Answer: **Piping** is a mechanism to connect output of one stream to another stream. It is normally used to get data from one stream and to pass output of that stream to another stream. There is no limit on piping operations.

Source: *tutorialspoint.com*

Q2: Name some of the events fired by streams. ☆☆☆☆

Answer: Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times. For example, some of the commonly used events are:

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

Source: *tutorialspoint.com*

Q3: What is the purpose of __filename variable? ☆☆☆☆

Answer: The `__filename` represents the filename of the code being executed. This is the resolved absolute path of this code file. For a main program this is not necessarily the same filename used in the command line. The value inside a module is the path to that module file.

Source: *tutorialspoint.com*

Q4: How can you listen on port 80 with Node? ☆☆☆☆

Answer: Run the application on any port above 1024, then put a reverse proxy like nginx in front of it.

Source: *blog.risingstack.com*

Q5: What tools can be used to assure consistent code style? ☆☆☆☆

Answer: You have plenty of options to do so:

- JSLint by Douglas Crockford
- JSHint
- ESLint
- JSCS

These tools are really helpful when developing code in teams, to enforce a given style guide and to catch common errors using static analysis.

Source: *blog.risingstack.com*

Q6: What's a stub? Name a use case. ☆☆☆☆

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

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

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

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

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

Source: *blog.risingstack.com*

Q7: Does Node.js support multi-core platforms? And is it capable of utilizing all the cores? ☆☆☆☆

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

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

Source: *techbeamers.com*

Q8: Is Node.js entirely based on a single-thread? ☆☆☆☆

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

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

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

Source: *techbeamers.com*

Q9: Is Node.js entirely based on a single-thread? ☆☆☆☆

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

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

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

Source: *techbeamers.com*

Q10: When to not use Node.js? ☆☆☆☆

Answer: We can use Node.js for a variety of applications. But it is a single threaded framework, so we should not use it for cases where the application requires long processing time. If the server is doing some calculation, it won’t be able to process any other requests. Hence, Node.js is best when processing needs less dedicated CPU time.

Source: *techbeamers.com*

Q11: Why to use Buffers instead of binary strings to handle binary data ? ☆☆☆☆

Answer: Pure JavaScript does not able to handle straight binary data very well. Since Node.js servers have to deal with TCP streams for reading and writing of data, binary strings will become problematic to work with as it is very slow and has a tendency to break. That's why it is always advisable to use Buffers instead of binary strings to handle binary data.

Source: *codingdefined.com*

Q12: How to use Buffer in Node.js? ☆☆☆

Answer: Buffer is used to process binary data, such as pictures, mp3, database files, etc. Buffer supports a variety of encoding and decoding, binary string conversion.

Source: *github.com/jimuyouyou*

Q13: When should I use EventEmitter? ☆☆☆

Answer: Whenever it makes sense for code to *subscribe* to something rather than get a callback from something. The typical use case would be that there's multiple blocks of code in your application that may need to do something when an event happens.

Source: *stackoverflow.com/*

Q14: How do you debug Node.js applications? ☆☆☆

Answer: Node has its own built in GUI debugger as of version 6.3 (using Chrome's DevTools).

```
node --inspect server.js
```

Some other options for debugging are:

- Joyent's Guide
- Debugger
- Node Inspector
- Visual Studio Code
- Cloud9
- Brackets

Source: *stackoverflow.com*

Q15: Rewrite promise-based Node.js applications to Async/Await ☆☆☆

Details: Rewrite this code to Async/Await:

```
function asyncTask() {
    return functionA()
        .then((valueA) => functionB(valueA))
        .then((valueB) => functionC(valueB))
        .then((valueC) => functionD(valueC))
        .catch((err) => logger.error(err))
}
```

Answer:

```
async function asyncTask() {
    try {
        const valueA = await functionA()
        const valueB = await functionB(valueA)
        const valueC = await functionC(valueB)
        return await functionD(valueC)
    } catch (err) {
        logger.error(err)
    }
}
```

Source: *stackoverflow.com*

Q16: What is the relationship between Node.js and V8? ☆☆☆

Answer: V8 is the Javascript engine inside of node.js that parses and runs your Javascript. The same V8 engine is used inside of Chrome to run javascript in the Chrome browser. Google open-sourced the V8 engine and the builders of node.js used it to run Javascript in node.js.

Source: *stackoverflow.com*

Q17: What is N-API in Node.js? ☆☆☆

Answer: N-API (pronounced N as in the letter, followed by API) is an API for building native Addons. It is independent from the underlying JavaScript runtime (ex V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate Addons from changes in the underlying JavaScript engine and allow modules compiled for one version to run on later versions of Node.js without recompilation.

Source: *medium.com*

Q18: Explain the concept of Domain in Node.js ☆☆☆

Answer: Domains provide a way to handle multiple different IO operations as a single group. If any of the event emitters or callbacks registered to a domain emit an `error` event, or throw an error, then the domain object will be notified, rather than losing the context of the error in the `process.on('uncaughtException')` handler, or causing the program to exit immediately with an error code.

Domain error handlers are not a substitute for closing down a process when an error occurs. The safest way to respond to a thrown error is to shut down the process. In a normal web server, the better approach is to send an error response to the request that triggered the error, while letting the others finish in their normal time, and stop listening for new requests in that worker.

```
var domain = require('domain');
var d = domain.create();
// Domain emits 'error' when it's given an unhandled error
d.on('error', function(err) {
    console.log(err.stack);
    // Our handler should deal with the error in an appropriate way
});

// Enter this domain
d.run(function() {
    // If an un-handled error originates from here, process.domain will handle it
    console.log(process.domain === d); // true
});

// domain has now exited. Any errors in code past this point will not be caught.
```

Source: *nodejs.org*

Q19: Are you familiar with differences between Node.js nodules and ES6 nodules? ☆☆☆

Answer: The modules used in Node.js follow a module specification known as the **CommonJS** specification. The recent updates to the JavaScript programming language, in the form of ES6, specify changes to the language, adding things like new class syntax and a module system. This module system is different from Node.js modules. To import ES6 module, we'd use the ES6 `import` functionality.

Now ES6 modules are incompatible with Node.js modules. This has to do with the way modules are loaded differently between the two formats. If you use a compiler like Babel, you can mix and match module formats.

Source: *stackoverflow.com*

Q20: What are the use cases for the Node.js "vm" core module? ☆☆☆

Answer: It can be used to safely execute a piece of code contained in a string or file. The execution is performed in a separate environment that by default has no access to the environment of the program that created it. Moreover, you can specify execution timeout and context-specific error handling.

Source: *quora.com*

Q1: What is Piping in Node? ☆☆☆☆

Answer: Piping is a mechanism to connect output of one stream to another stream. It is normally used to get data from one stream and to pass output of that stream to another stream. There is no limit on piping operations.

Source: *tutorialspoint.com*

Q2: Name some of the events fired by streams. ☆☆☆☆

Answer: Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times. For example, some of the commonly used events are:

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

Source: *tutorialspoint.com*

Q3: What is the purpose of __filename variable? ☆☆☆☆

Answer: The `__filename` represents the filename of the code being executed. This is the resolved absolute path of this code file. For a main program this is not necessarily the same filename used in the command line. The value inside a module is the path to that module file.

Source: *tutorialspoint.com*

Q4: How can you listen on port 80 with Node? ☆☆☆☆

Answer: Run the application on any port above 1024, then put a reverse proxy like nginx in front of it.

Source: *blog.risingstack.com*

Q5: What tools can be used to assure consistent code style? ☆☆☆☆

Answer: You have plenty of options to do so:

- JSLint by Douglas Crockford
- JSHint
- ESLint
- JSCS

These tools are really helpful when developing code in teams, to enforce a given style guide and to catch common errors using static analysis.

Source: *blog.risingstack.com*

Q6: What's a stub? Name a use case. ☆☆☆☆

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

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

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

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

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

Source: *blog.risingstack.com*

Q7: Does Node.js support multi-core platforms? And is it capable of utilizing all the cores? ☆☆☆☆

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

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

Source: *techbeamers.com*

Q8: Is Node.js entirely based on a single-thread? ☆☆☆☆

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

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

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

Source: *techbeamers.com*

Q9: Is Node.js entirely based on a single-thread? ☆☆☆☆

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

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

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

Source: *techbeamers.com*

Q10: When to not use Node.js? ☆☆☆☆

Answer: We can use Node.js for a variety of applications. But it is a single threaded framework, so we should not use it for cases where the application requires long processing time. If the server is doing some calculation, it won’t be able to process any other requests. Hence, Node.js is best when processing needs less dedicated CPU time.

Source: *techbeamers.com*

Q11: Why to use Buffers instead of binary strings to handle binary data ? ☆☆☆☆

Answer: Pure JavaScript does not able to handle straight binary data very well. Since Node.js servers have to deal with TCP streams for reading and writing of data, binary strings will become problematic to work with as it is very slow and has a tendency to break. That's why it is always advisable to use Buffers instead of binary strings to handle binary data.

Source: *codingdefined.com*

Q12: How to gracefully Shutdown Node.js Server? ☆☆☆☆

Answer: We can gracefully shutdown Node.js server by using the generic signal called SIGTERM or SIGINT which is used for program termination. We need to call SIGTERM or SIGINT which will terminate the program and clean up the resources utilized by the program.

Source: *codingdefined.com*

Q13: What are the timing features of Node.js? ☆☆☆☆

Answer: The Timers module in Node.js contains functions that execute code after a set period of time.

- **setTimeout/clearTimeout** - can be used to schedule code execution after a designated amount of milliseconds
- **setInterval/clearInterval** - can be used to execute a block of code multiple times
- **setImmediate/clearImmediate** - will execute code at the end of the current event loop cycle
- **process.nextTick** - used to schedule a callback function to be invoked in the next iteration of the Event Loop

```
function cb(){
  console.log('Processed in next iteration');
}
process.nextTick(cb);
console.log('Processed in the first iteration');
```

Output:

```
Processed in the first iteration
Processed in next iteration
```

Source: *github.com/jimuyouyou*

Q14: Explain usage of NODE_ENV ☆☆☆☆

Answer: Node encourages the convention of using a variable called NODE_ENV to flag whether we're in production right now. This determination allows components to provide better diagnostics during development, for example by disabling caching or emitting verbose log statements. Setting NODE_ENV to production makes your application 3 times faster.

```
// Setting environment variables in bash before starting the node process
$ NODE_ENV=development
$ node

// Reading the environment variable using code
if (process.env.NODE_ENV === “production”)
    useCaching = true;
```

Source: *github.com/i0natan/nodebestpractices*

Q15: What is LTS releases of Node.js why should you care? ☆☆☆☆

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

LTS versions of Node.js are supported for at least 18 months and are indicated by even version numbers (e.g. 4, 6, 8). They're best for production since the LTS release line is focussed on stability and security, whereas the *Current* release line has a shorter lifespan and more frequent updates to the code. Changes to LTS versions are limited to bug fixes for stability, security updates, possible npm updates, documentation updates and certain performance improvements that can be demonstrated to not break existing applications.

Source: *github.com/i0natan/nodebestpractices*

Q16: Provide some example of config file separation for dev and prod environments ☆☆☆☆

Answer: A perfect and flawless configuration setup should ensure:

- keys can be read from file AND from environment variable
- secrets are kept outside committed code
- config is hierarchical for easier findability

Consider the following config file:

```
var config = {
  production: {
    mongo : {
      billing: '****'
    }
  },
  default: {
    mongo : {
      billing: '****'
    }
  }
}

exports.get = function get(env) {
  return config[env] || config.default;
}
```

And it's usage:

```
const config = require('./config/config.js').get(process.env.NODE_ENV);
const dbconn = mongoose.createConnection(config.mongo.billing);
```

Source: *github.com/i0natan/nodebestpractices*

Q17: How would you handle errors for async code in Node.js? ☆☆☆☆

Answer: Handling async errors in callback style (error-first approach) is probably the fastest way to hell (a.k.a the pyramid of doom). It's better to use a reputable promise library or async-await instead which enables a much more compact and familiar code syntax like try-catch.

Consider promises to catch errors:

```
doWork()
 .then(doWork)
 .then(doOtherWork)
 .then((result) => doWork)
 .catch((error) => {throw error;})
 .then(verify);
```

or using async/await:

```
async function check(req, res) {
    try {
        const a = await someOtherFunction();
        const b = await somethingElseFunction();
        res.send("result")
    } catch (error) {
        res.send(error.stack);
    }
}
```

Source: *github.com/i0natan/nodebestpractices*

Q18: What's the difference between dependencies, devDependencies and peerDependencies in npm package.json file? ☆☆☆☆

Answer:

- **dependencies** - Dependencies that your project needs to run, like a library that provides functions that you call from your code. They are installed transitively (if A depends on B depends on C, npm install on A will install B and C).
- **devDependencies** - Dependencies you only need during development or releasing, like compilers that take your code and compile it into javascript, test frameworks or documentation generators. They are not installed transitively (if A depends on B dev-depends on C, npm install on A will install B only).

- **peerDependencies** - Dependencies that your project hooks into, or modifies, in the parent project, usually a plugin for some other library or tool. It is just intended to be a check, making sure that the parent project (project that will depend on your project) has a dependency on the project you hook into. So if you make a plugin C that adds functionality to library B, then someone making a project A will need to have a dependency on B if they have a dependency on C. They are not installed (unless npm < 3), they are only checked for.

Source: *stackoverflow.com*

Q19: How do you convert an existing callback API to promises? ☆☆☆☆

Details: How to convert this callback code to Promise? Provide some examples.

```
function divisionAPI (number, divider, successCallback, errorCallback) {
    if (divider == 0) {
        return errorCallback( new Error("Division by zero") )
    }
    successCallback( number / divider )
}
```

Answer:

```
function divisionAPI(number, divider) {
    return new Promise(function(fulfilled, rejected) {
        if (divider == 0) {
            return rejected(new Error("Division by zero"))
        }
        fulfilled(number / divider)
    })
}

// Promise can be used with together async\await in ES7 to make the program flow wait f
async function foo() {
    var result = await divisionAPI(1, 2); // awaits for a fulfilled result!
    console.log(result);
}

// Another usage with the same code by using .then() method
divisionAPI(1, 2).then(function(result) {
    console.log(result)
})
```

Node.js 8.0.0 includes a new `util.promisify()` API that allows standard Node.js callback style APIs to be wrapped in a function that returns a Promise.

```
const fs = require('fs');
const util = require('util');

const readfile = util.promisify(fs.readFile);

readfile('/some/file')
    .then((data) => {
        /** ... **/
     })
    .catch((err) => {
        /** ... **/
    });
```

Source: *stackoverflow.com*

Q20: What are async functions in Node? Provide some examples. ☆☆☆☆

Answer: With the release of Node.js 8, the long awaited async functions have landed in Node.js as well. ES 2017 introduced Asynchronous functions. Async functions are essentially a cleaner way to work with asynchronous code in JavaScript.

Async/Await is:

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

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

Q1: Consider following code snippet ☆☆☆☆☆

Details: Consider following code snippet:

```
{
  console.time("loop");
  for (var i = 0; i < 1000000; i += 1) {
    // Do nothing
  }
  console.timeEnd("loop");
}
```

The time required to run this code in Google Chrome is considerably more than the time required to run it in Node.js Explain why this is so, even though both use the v8 JavaScript Engine.

Answer: Within a web browser such as Chrome, declaring the variable `i` outside of any function’s scope makes it global and therefore binds it as a property of the `window` object. As a result, running this code in a web browser requires repeatedly resolving the property `i` within the heavily populated `window` namespace in each iteration of the `for` loop.

In Node.js, however, declaring any variable outside of any function’s scope binds it only to the module’s own scope (not the `window` object) which therefore makes it much easier and faster to resolve.

Source: *toptal.com*

Q2: Can Node.js use other engines than V8? ☆☆☆☆☆

Answer: Yes. Microsoft Chakra is another JavaScript engine which can be used with Node.js. It’s not officially declared yet.

Source: *codeforgeek.com*

Q3: How would you scale Node application? ☆☆☆☆☆

Answer: We can scale Node application in following ways:

- cloning using *Cluster* module.
- decomposing the application into smaller services – i.e micro services.

Source: *codeforgeek.com*

Q4: What is the difference between process.nextTick() and setImmediate() ? ☆☆☆☆☆

Answer: The difference between `process.nextTick()` and `setImmediate()` is that `process.nextTick()` defers the execution of an action till the next pass around the event loop or it simply calls the callback function once the ongoing execution of the event loop is finished whereas `setImmediate()` executes a callback on the next cycle of the event loop and it gives back to the event loop for executing any I/O operations.

Source: *codingdefined.com*

Q5: How to solve "Process out of Memory Exception" in Node.js ? ☆☆☆☆☆

Answer: To solve the process out of memory exception in Node.js we need to increase the `max-old-space-size`. By default the max size of max-old-space-size is 512 mb which you can increase by the command:

```
node --max-old-space-size=1024 file.js
```

Source: *codingdefined.com*

Q6: Explain what is Reactor Pattern in Node.js? ☆☆☆☆☆

Answer: Reactor Pattern is an idea of non-blocking I/O operations in Node.js. This pattern provides a handler(in case of Node.js, a *callback function*) that is associated with each I/O operation. When an I/O request is generated, it is submitted to a *demultiplexer*.

This *demultiplexer* is a notification interface that is used to handle concurrency in non-blocking I/O mode and collects every request in form of an event and queues each event in a queue. Thus, the demultiplexer provides the *Event Queue*.

At the same time, there is an Event Loop which iterates over the items in the Event Queue. Every event has a callback function associated with it, and that callback function is invoked when the Event Loop iterates.

Source: *hackernoon.com*

Q7: Explain some Error Handling approaches in Node.js you know about. Which one will you use? ☆☆☆☆☆

Answer: Error handling in an asynchronous language works in a unique way and presents many challenges, some unexpected. There are four main error handling patterns in node:

- **Error return value** - doesn't work asynchronously

```
var validateObject = function (obj) {
    if (typeof obj !== 'object') {
        return new Error('Invalid object');
    }
};
```

- **Error throwing** - well-establish pattern, in which a function does its thing and if an error situation arises, it simply bails out throwing an error. Can leave you in an unstable state. It requires extra work to catch them. Also wrapping the async calls in try/catch won't help because the errors happen asynchronously. To fix this, we need *domains*. Domains provide an asynchronous try...catch.

```
var validateObject = function (obj) {
    if (typeof obj !== 'object') {
        throw new Error('Invalid object');
    }
};

try {
    validateObject('123');
}
catch (err) {
    console.log('Thrown: ' + err.message);
}
```

- **Error callback** - returning an error via a callback is the most common error handling pattern in Node.js. Handling error callbacks can become a mess (callback hell or the pyramid of doom).

```
var validateObject = function (obj, callback) {
    if (typeof obj !== 'object') {
        return callback(new Error('Invalid object'));
    }
    return callback();
};

validateObject('123', function (err) {
    console.log('Callback: ' + err.message);
});
```

- **Error emitting** - when emitting errors, the errors are broadcast to any interested subscribers and handled within the same process tick, in the order subscribed.

```
var Events = require('events');
var emitter = new Events.EventEmitter();

var validateObject = function (a) {
    if (typeof a !== 'object') {
        emitter.emit('error', new Error('Invalid object'));
    }
};

emitter.on('error', function (err) {
    console.log('Emitted: ' + err.message);
});

validateObject('123');
```

- **Promises** for async error handling

```
doWork()
.then(doWork)
.then(doError)
.then(doWork)
.catch(errorHandler)
.then(verify);
```

- **Try...catch with async/await** - ES7 Async/await allows us as developers to write asynchronous JS code that look synchronous.

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

- **Await-to-js lib** - async/await without try-catch blocks in Javascript

```
import to from 'await-to-js';

async function main(callback) {
    const [err,quote] = await to(getQuote());

    if(err || !quote) return callback(new Error('No Quote found');

    callback(null,quote);

}
```

Source: *gist.github.com*

Q8: Why should you separate Express 'app' and 'server'? ☆☆☆☆☆

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

API declaration, should reside in app.js:

```
var app = express();
app.use(bodyParser.json());
app.use("/api/events", events.API);
app.use("/api/forms", forms);
```

Server network declaration, should reside in /bin/www:

```
var app = require('../app');
var http = require('http');

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
 * Create HTTP server.
 */

var server = http.createServer(app);
```

Source: *github.com/i0natan/nodebestpractices*

Q9: Rewrite the code sample without try/catch block ☆☆☆☆☆

Details: Consider the code:

```
async function check(req, res) {
  try {
    const a = await someOtherFunction();
    const b = await somethingElseFunction();
    res.send("result")
  } catch (error) {
    res.send(error.stack);
  }
}
```

Rewrite the code sample without try/catch block.

Answer:

```
async function getData(){
  const a = await someFunction().catch((error)=>console.log(error));
  const b = await someOtherFunction().catch((error)=>console.log(error));
```

```
    if (a && b) console.log("some result")
  }
```

or if you wish to know which specific function caused error:

```
  async function loginController() {
    try {
      const a = await loginService().
      catch((error) => {
        throw new CustomErrorHandler({
          code: 101,
          message: "a failed",
          error: error
        })
      });
      const b = await someUtil().
      catch((error) => {
        throw new CustomErrorHandler({
          code: 102,
          message: "b failed",
          error: error
        })
      });
      //someoeeoe
      if (a && b) console.log("no one failed")
    } catch (error) {
      if (!(error instanceof CustomErrorHandler)) {
        console.log("gen error", error)
      }
    }
  }
```

Source: *medium.com*

Q10: How many threads does Node actually create? ☆☆☆☆☆

Answer: 4 extra threads are for use by V8. V8 uses these threads to perform various tasks, such as GC-related background tasks and optimizing compiler tasks.

Source: *stackoverflow.com*

Q11: Can Node.js work without V8? ☆☆☆☆

Answer: No. The current node.js binary cannot work without V8. It would have no Javascript engine and thus no ability to run code which would obviously render it non-functional. Node.js was not designed to run with any other Javascript engine and, in fact, all the native code bindings that come with node.js (such as the fs module or the net module) all rely on the specific V8 interface between C++ and Javascript.

There is an effort by Microsoft to allow the Chakra Javascript engine (that's the engine in Edge) to be used with node.js. Node.js can actually function to some extent without V8, through use of the node-chakracore project. There is ongoing work to reduce the tight coupling between V8 and Node, so that different JavaScript engines can be used in-place.

Source: *stackoverflow.com*

Q12: How the V8 engine works? ☆☆☆☆

Answer: **V8** is a JavaScript engine built at the google development center, in Germany. It is open source and written in C++. It is used for both client side (Google Chrome) and server side (node.js) JavaScript applications.

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

Source: *nodejs.org*

Q13: What is the purpose of using hidden classes in V8? ☆☆☆☆☆

Answer: JavaScript is a prototype-based language: there are no classes and objects are created by using a cloning process. JavaScript is also dynamically typed: types and type informations are not explicit and properties can be added to and deleted from objects on the fly.

Accessing types and properties effectively makes a first big challenge for V8. Instead of using a dictionary-like data structure for storing object properties and doing a dynamic lookup to resolve the property location (like most JavaScript engines do), V8 creates** hidden classes**, at runtime, in order to have an internal representation of the type system and to improve the property access time.

Source: *thibaultlaurens.github.io*

Q14: How V8 compiles JavaScript code? ☆☆☆☆☆

Answer: V8 has two compilers:

- A **“Full” Compiler** that can generate good code for any JavaScript: good but not great JIT code. The goal of this compiler is to generate code quickly. To achieve its goal, it doesn’t do any type analysis and doesn’t know anything about types. Instead, it uses an Inline Caches or “IC” strategy to refine knowledge about types while the program runs. IC is very efficient and brings about 20 times speed improvment.

- An **Optimizing Compiler** that produces great code for most of the JavaScript language. It comes later and re-compiles hot functions. The optimizing compiler takes types from the Inline Cache and make decisions about how to optimize the code better. However, some language features are not supported yet like try/catch blocks for instance. (The workaround for try/catch blocks is to write the “non stable” code in a function and call the function in the try block)

V8 also supports **de-optimization**: the optimizing compiler makes optimistic assumptions from the Inline Cache about the different types, de-optimization comes if these assumptions are invalid. For example, if a hidden class generated was not the one expected, V8 throws away the optimized code and comes back to the Full Compiler to get types again from the Inline Cache. This process is slow and should be avoided by trying to not change functions after they are optimized.

Source: *thibaultlaurens.github.io*

Q15: How does libuv work under the hood? ☆☆☆☆☆

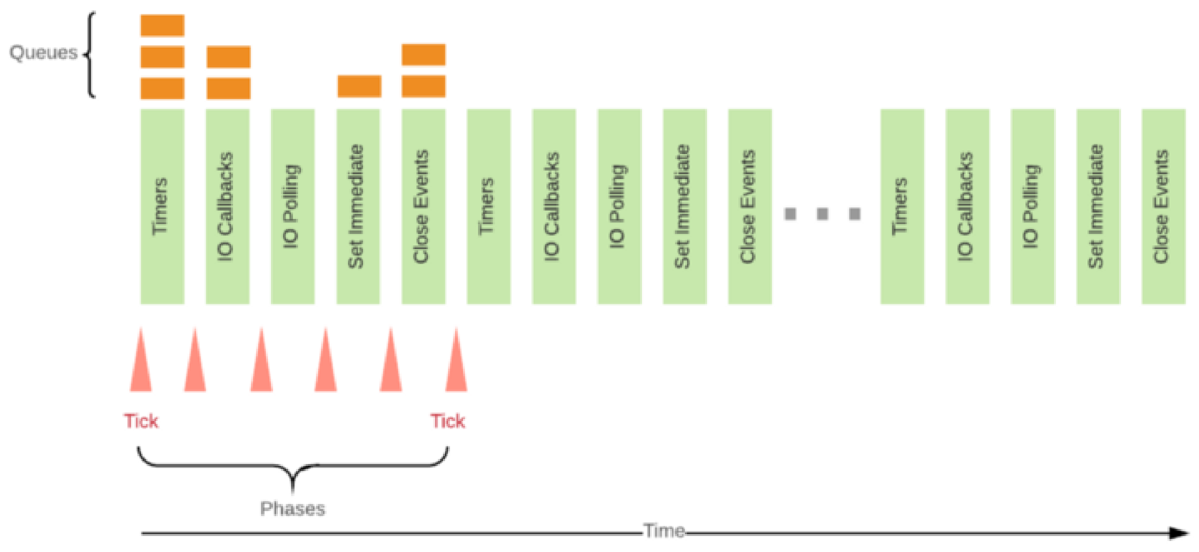
Answer: There is only one thread that executes JavaScript code and this is the thread where the event loop is running provided by **libuv**. The execution of callbacks (know that every userland code in a running Node.js application is a callback) is done by the event loop.

Libuv by default creates a thread pool with four threads to offload asynchronous work to. Today’s operating systems already provide asynchronous interfaces for many I/O tasks (e.g. AIO on Linux). Whenever possible, libuv will use those asynchronous interfaces, avoiding usage of the thread pool.

The event loop as a process is a set of phases with specific tasks that are processed in a round-robin manner. Each phase has a FIFO queue of callbacks to execute. While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

- **timers**: this phase executes callbacks scheduled by `setTimeout()` and `setInterval()` .
- **pending callbacks**: executes I/O callbacks deferred to the next loop iteration.
- **idle, prepare**: only used internally.

- **poll**: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.
- **check**: `setImmediate()` callbacks are invoked here.
- **close callbacks**: some close callbacks, e.g. `socket.on('close', ...)` .



Source: *nodejs.org*

Q16: How does the cluster module work? What's the difference between it and a load balancer? ☆☆☆☆

Answer: The cluster module performs fork from your server (at that moment it is already an OS process), thus creating several slave processes. The cluster module supports two methods of distributing incoming connections.

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

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

The difference between a cluster module and a load balancer is that instead of distributing load between processes, the balancer distributes requests.

Source: *imasters.com*

Q17: What is V8 Templates? ☆☆☆☆☆

Answer: A template is a blueprint for JavaScript functions and objects. You can use a template to wrap C++ functions and data structures within JavaScript objects. V8 has two types of templates: Function Templates and Object Templates.

- **Function Template** is the blueprint for a single function. You create a JavaScript instance of template by calling the template’s GetFunction method from within the context in which you wish to instantiate the JavaScript function. You can also associate a C++ callback with a function template which is called when the JavaScript function instance is invoked.
- **Object Template** is used to configure objects created with function template as their constructor. You can associate two types of C++ callbacks with object templates: accessor callback and interceptor callback. Accessor callback is invoked when a specific object property is accessed by a script. Interceptor callback is invoked when any object property is accessed by a script. In a nutshell, you can wrap C++ objects\structures within JavaScript objects.

Source: *blog.ghaiklor.com*

Q18: Why do we need C++ Addons in Node.js? ☆☆☆☆☆

Answer: Node.js Addons are dynamically-linked shared objects, written in C++, that can be loaded into Node.js using the require() function, and used just as if they were an ordinary Node.js module. They are used primarily to provide an interface between JavaScript running in Node.js and C/C++ libraries.

There can be many reasons to write nodejs addons:

1. You may want to access some native apis that is difficult using JS alone.
2. You may want to integrate a third party library written in C/C++ and use it directly in Node.js.
3. You may want to rewrite some of the modules in C++ for performance reasons.

N-API (pronounced N as in the letter, followed by API) is an API for building native Addons.

Source: *nodejs.org*

Q19: Is it possible to use "Class" in Node.js? ☆☆☆☆

Answer: With ES6, you are able to make "actual" classes just like this:

```
class Animal {

    constructor(name) {
        this.name = name;
    }
    print() {
        console.log('Name is :' + this.name);
    }
}
```

You can export a class just like anything else:

```
module.exports = class Animal {

};
```

Once imported into another module, then you can treat it as if it were defined in that file:

```
var Animal = require('./Animal');

class Cat extends Animal {
    ...
}
```

Source: *stackoverflow.com*

Q20: Why Node.js devs tend to lean towards the Module Requiring vs Dependency Injection? ☆☆☆☆☆

Answer: Dependency injection is somewhat the opposite of normal *module design*. In normal module design, a module uses `require()` to load in all the other modules that it needs with the goal of making it simple for the caller to use your module. The caller can just require() in your module and your module will load all the other things it needs.

With dependency injection, rather than the module loading the things it needs, the caller is required to pass in things (usually objects) that the module needs. This can make certain types of testing easier and it can make mocking certain things for testing purposes easier.

Modules and dependency injection are orthogonal: if you need dependency injection for testability or extensibility then use it. If not, importing modules is fine. The great thing about JS is that you can modify just about anything to achieve what you want. This comes in handy when it comes to testing.

Source: *reddit.com*

Q1: Explain the result of this code execution ☆☆☆☆☆

Details: Explain the result of that code execution:

```
var EventEmitter = require("events");

var crazy = new EventEmitter();

crazy.on('event1', function () {
    console.log('event1 fired!');
    crazy.emit('event2');
});

crazy.on('event2', function () {
    console.log('event2 fired!');
    crazy.emit('event3');

});

crazy.on('event3', function () {
    console.log('event3 fired!');
    crazy.emit('event1');
});

crazy.emit('event1');
```

Answer: You'll get an exception that basically says the call stack has exploded. Why? Every emit will invoke synchronous code. Because all callbacks are executed in a synchronous manner it'll just recursive call itself to infinity and beyond.

Output:

```
console.js:165
    if (isStackOverflowError(e))
        ^

RangeError: Maximum call stack size exceeded
    at write (console.js:165:9)
    at Console.log (console.js:197:3)
    at EventEmitter.<anonymous> (C:\Work\Node\main.js:6:13)
    at EventEmitter.emit (events.js:182:13)
    at EventEmitter.<anonymous> (C:\Work\Node\main.js:18:11)
    at EventEmitter.emit (events.js:182:13)
    at EventEmitter.<anonymous> (C:\Work\Node\main.js:12:11)
    at EventEmitter.emit (events.js:182:13)
    at EventEmitter.<anonymous> (C:\Work\Node\main.js:7:11)
    at EventEmitter.emit (events.js:182:13)
```

Source: *codementor.io*

Q2: Explain the result of this code execution ☆☆☆☆☆

Details: Explain the result of this code execution

```
var EventEmitter = require('events');

var crazy = new EventEmitter();

crazy.on('event1', function () {
    console.log('event1 fired!');
    setImmediate(function () {
        crazy.emit('event2');
    });
});

crazy.on('event2', function () {
    console.log('event2 fired!');
    setImmediate(function () {
        crazy.emit('event3');
    });

});

crazy.on('event3', function () {
    console.log('event3 fired!');
    setImmediate(function () {
        crazy.emit('event1');
    });
});

crazy.emit('event1');
```

Answer: Shortly - the app will be run infinitely. Any function passed as the setImmediate() argument is a callback that's executed in the *next iteration* of the event loop.
Without `setImmidiate` all callbacks are executed in a synchronous manner.
With `setImmidiate` each call back executed as a part of next event loop iteration so no recursion/stuck occurs.

Source: *codementor.io*

Q3: What will happen when that code will be executed? ☆☆☆☆☆

Details:

What will happen when that code will be executed?

```
var EventEmitter = require('events');

var crazy = new EventEmitter();

crazy.on('event1', function () {
    console.log('event1 fired!');
    process.nextTick(function () {
        crazy.emit('event2');
    });
});

crazy.on('event2', function () {
    console.log('event2 fired!');
    process.nextTick(function () {
        crazy.emit('event3');
    });

});

crazy.on('event3', function () {
    console.log('event3 fired!');
    process.nextTick(function () {
        crazy.emit('event1');
    });
});

crazy.emit('event1');
```

Answer: It'll get stuck! And if you wait long enough, about 30 seconds, it'll eventually give you a "process out of memory" exception. Now, the problem is not stack overflow, it's GC not being able to reclaim memory. Every handler has its own closure to access the crazy on the outer layer. This cost comes out of the heap. Though you might not be 100% why GC can't successfully get the memory back, you can probably guess that the program got stuck in some even loop phase because there's always another `process.nextTick` callback to be processed. So essentially, the event loop is blocked completely.

Source: _codementor.io_Consider the code:

```
function doubleAfter2Seconds(x) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(x * 2);
        }, 2000);
    });
}
```

What if we want to run a few different values through our function and add the result?

Promise-based solution will be:

```
function addPromise(x) {
    return new Promise(resolve => {
        doubleAfter2Seconds(10).then((a) => {
            doubleAfter2Seconds(20).then((b) => {
                doubleAfter2Seconds(30).then((c) => {
                    resolve(x + a + b + c);
                })
            })
        })
    });
}

addPromise(10).then((sum) => {
  console.log(sum);
});
```

Async/Await solution will look like:

```
async function addAsync(x) {
    const a = await doubleAfter2Seconds(10);
    const b = await doubleAfter2Seconds(20);
    const c = await doubleAfter2Seconds(30);
    return x + a + b + c;
}

addAsync(10).then((sum) => {
  console.log(sum);
});
```

Source: *medium.com*