Class 14 - MERN Interview questions

Agenda

1. Nodejs modules
    a. Inbuilt modules
        i. Inbuilt modules are integrated into Node.js and can be used without any additional installation. They provide core functionalities like file system access, HTTP server creation, path manipulation, and more.
        ii. Examples:
        iii. http: For creating HTTP servers.
        iv. fs: For handling file system operations.
        v. path: For working with file and directory paths.
    b. Local modules
        i. Local modules are custom modules created in a project. They help in structuring and organizing code into different files and directories.
        ii. Creating & Using: You can create a local module by exporting functions, objects, or variables from a file using module.exports and then require them in other files as needed.
    c. Third party modules
        i. Third-party modules are libraries or packages developed by the community, available through npm (Node Package Manager).

    ii. Usage: They are installed using npm install <package_name> and included in your project using require().
    iii.
2. Nodejs streams
    a. Streams are collections of data that might not be available all at once and don't have to fit in memory. They allow handling of reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way
    b. Like in video streams, A portion of the video is buffered (pre-loaded) and played while the next portion is being downloaded. This approach ensures a smooth viewing experience without requiring the entire file to be downloaded first.
3. Doing process heavy tasks
    a. Node.js is single-threaded, and CPU-intensive tasks can block the event loop, affecting performance.
    b. Child Processes: Offload heavy computation to child processes using the child_process module.

Nodejs

1. JavaScript on the Server: Traditionally, JavaScript was used only in web browsers. Node.js allows you to use JavaScript to write

server-side code, meaning you can write the logic that runs on your server using JavaScript.
2. Runtime Environment: It's not a programming language or a framework, but a runtime environment that allows JavaScript to be run on the server side.
3. Built on Chrome's V8 Engine: Node.js runs on the V8 JavaScript engine, which powers Google Chrome. This means it's incredibly fast and efficient at running JavaScript code.

Main features

1. Node.js uses non-blocking, event-driven architecture. It's designed to handle asynchronous operations, allowing it to manage multiple operations concurrently without waiting for any to complete.
2. This makes it very efficient for tasks like reading/writing to the file system, network operations, or any operations that rely on external data sources.
3. Single-Threaded: Despite being single-threaded, it can handle numerous concurrent connections efficiently due to its event-driven nature and use of callbacks or promises.
4. Non-Blocking:
    a. Node.js doesn’t wait for tasks like reading files or database queries to finish before moving on to the next task. It starts a task, and while it's being processed, Node.js can start handling another task.

5. Event-Driven:
    a. In Node.js: When an event occurs, Node.js reacts by executing the code meant to handle that event (like processing the file or responding to the user's request).
6. When you write a server in Node.js and a user sends a request (like asking for a webpage), Node.js takes this request and processes it. While processing (like fetching data from a database), Node.js doesn’t just wait around; it can take more requests and start working on them.
7. Once the data for the first request is ready (the event), Node.js finishes processing that request (sends back the webpage) and then moves on to complete other tasks

Browsers and NodeJS

1. Both use Google's V8 JavaScript engine to compile JavaScript into native machine code. This engine is known for its performance and efficiency.
2. Both support JavaScript, meaning the core syntax and standard JavaScript functions are the same across both platforms.
3. Browsers: Provide a runtime environment for client-side JavaScript, enabling interaction with web pages (DOM manipulation), handling user events, and rendering content.
4. Node.js: Provides a server-side runtime environment. It extends JavaScript capabilities to interact with the filesystem, perform

network operations, and run applications outside of a browser context.

5. APIs and Global Objects:
    a. Browsers: Offer Web APIs like DOM, WebRTC, Fetch API for HTTP requests, and more are available through global window object
    b. Node.js: Provides its own set of APIs for server-side operations, like file system manipulation (fs module), creating HTTP servers (http module), etc. Global objects like global and process-specific objects like process are unique to Node.js.

Availability of APIs

1. Browser Environment and APIs:
    a. Directly Available: In a web browser, APIs like the DOM, fetch, and others are directly available as part of the browser's global environment. You don't need to import them using require or any other import mechanism.
    b. Global Window Object: These APIs are typically attached to the window object, which represents the global scope in the browser. For example, fetch is accessible as window.fetch, and similarly, the document object for the DOM is window.document.
    c. No Require Statement Needed: Since these APIs are built into the browser, they are automatically loaded and ready to

use in any webpage's JavaScript without needing explicit import statements.

2. Node.js Environment and APIs:
    a. Module System: Node.js uses a module system (CommonJS, specifically). Most of its core functionality, like file system (fs), networking (http), path operations (path), etc., are organized as modules.
    b. Using Require: To use these modules, you need to import them into your script using the require function. This is because Node.js does not automatically load all modules to keep the global namespace clean and to optimize performance.

Code

1. Create a new folder and a new file (say nodeExplorer.js)

```
console.log(global)
console.log("dir name",__dirname,"file name",__filename)
```

2. Observe the methods like clearTimeout, setInterval etc are part og global object. Notice the directory name and file name
3. The other features like fs, path, http etc are modules and need to be included in the code
4. Try doing console.log(process)

a. process is a global object that provides information about, and control over, the current Node.js process. It is one of the core modules and is available without needing to import it using require.
b. Environment Variables (process.env):
    i. Stores environment variables as key-value pairs. It's commonly used to access system environment variables or set configuration options for the application
c. Current Working Directory (process.cwd()):
    i. Returns the current working directory of the Node.js process.
d. Command Line Arguments (process.argv):
    i. An array containing the command-line arguments passed when the Node.js process was launched.
    ii. node app.js arg1 arg2 will result in process.argv being ['path/to/node', 'path/to/app.js', 'arg1', 'arg2'].
e. Process ID (process.pid):
    i. The process ID of the Node.js process.
f. Standard Input/Output (process.stdin, process.stdout, process.stderr):
    i. Streams for interacting with input/output. For example, process.stdout is used to write output to the terminal.
g. process.moduleLoadList

i. This array contains the names of the built-in modules that have been loaded by the process. It's useful for debugging or understanding which modules your application is using.

ii. Notice the internal bindings and native modules

    1. "Internal Binding" modules refer to the internal C++ bindings that are used by Node.js. These bindings are essentially the lower-level code that Node.js uses to interact with the underlying system and V8 engine.
    2. they provide functionality that is not directly exposed to the Node.js user but is used internally by various Node.js core modules. For example, bindings to the file system, network, or other system-level operations.
    3. NativeModule" refers to modules that are written in JavaScript but are part of the Node.js core. They are "native" in the sense that they come bundled with Node.js itself.
    4. Unlike internal bindings, NativeModules are often directly accessible and usable in Node.js applications. They form the standard library of Node.js.

Some awesome resources

1. https://github.com/sindresorhus/awesome-nodejs
    a. This repository is a valuable resource for anyone looking to explore the Node.js ecosystem, find useful packages, or learn more about Node.js development.
2. https://github.com/enaqx/awesome-react

Problem statement 1

1. Copy a large file in the folder
2. If you want to generate large files with code

```
// // Generate random content
// const content =
Math.random().toString(36).repeat(10000000); //
Approximately 130MB

// // Write content to file
//
fs.writeFileSync('/Users/scaler/Documents/BEAug16/Nov-22/no
deDiscussion/big.file', content);
```

3. When a user requests for a large file, how do i serve them
4. Problem statement : How can a Node.js application effectively serve large files, such as a 400MB video for eg, without exhausting the server's RAM and cache resources?
5. Create a basic server using http module

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

```
const server = http.createServer();


server.listen(3000, () => {
   console.log("Server started at 3000")
})
```

6. Adding a listener for request event when someone sends a request to our server

```
Const fs = require('fs')
server.on('request', (req, res) => {
   fs.readFile('./big.file', (err, data) => {
       if(err) throw err;
       res.end(data);
   })
})
```

7. ,the request event is handled using a framework like Express.js, which abstracts the low-level handling of HTTP requests. For now we are simplifying this for our problem statement

8. Open the activity monitor and a terminal
9. In the terminal do curl curl http://localhost:3000
10. Notice the memory jump in the terminal because the file was first brought in the memory and then served
11. Solution -> streaming

Problem Statement 2

1. * there can be certain tasks that are CPU intensive like image processing, video encoding, etc.
2. We will take example of fibonacci computation
3. Create another file cpu.js

```
/**
* there can be certain tasks that are CPU intensive like
image processing,
* video encoding, etc.
*/
const express = require('express');
const cors = require('cors');
const app = express();
// serial and parallel
function calculateFibonacci(number) {
   if (number <= 1) {
       return number;
   }
   return calculateFibonacci(number - 1) +
calculateFibonacci(number - 2);
}
app.use(cors());
app.get('/fib', (req, res) => {
   const { number, requestNumber } = req.query;
   console.log("handler fn ran for req", requestNumber);
   if (!number || isNaN(number) || number <= 0) {
       return res.status(400).json({ error: 'Please provide
a valid positive number.' });
   }
   const answer = calculateFibonacci(number);
```

```
    // console.log(answer);
    res.status(200).json({
        status: "success",
        message: answer,
        requestNumber
    })
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

4. For I/O operations (like reading from a file or database), Node.js uses non-blocking, asynchronous calls, allowing the event loop to continue handling other tasks while waiting for the I/O operation to complete.
5. However, CPU-bound tasks like Fibonacci computation don't yield control back to the event loop until they are fully completed.
6. This leads to a situation where the server becomes unresponsive or significantly slow in handling new requests.
7. Create a cpu.html file and have this code

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Fetch Request on Button Click</title>
</head>
```

```
<body>
    <h1>Fetch Request on Button Click</h1>
    <button id="fetchButton">Fetch Data</button>
    <script>
        console.log("Fetching data...");
        let requestNumber = 0
        function fetchData() {
            console.log("sending request", requestNumber)
            // console.log("Fetching data...");
            fetch('http://localhost:3000/fib?number=' + 40 +
'&requestNumber=' + requestNumber++)
                .then(response => response.json())
                .then(data => {
                    console.log('Response:', data);
                })
                .catch(error => {
                    console.error('Error fetching data:',
error);
                });
        }


document.getElementById('fetchButton').addEventListener('cl
ick', function () {

            const COuntId = setInterval(fetchData, 100);
            setTimeout(() => {
                clearInterval(COuntId)
            }, 5000);
        });
```

```
    </script>
</body>


</html>
```

8. We will see in the console tab that the response is delayed because of the computations that are involved

Understanding role of libuv in request handling

1. libuv: It's a cross-platform C library that Node.js uses under the hood. It provides asynchronous I/O capabilities, handling operations like file system operations, networking, and timers.
2. Request Flow:
    a. When a request arrives, it's initially handled by the operating systems (networking layer).
    b. libuv then picks up the request from the OS and queues it for processing by Node.js.
3. Handling Overwhelmed Server: If the Node.js server is busy (e.g., executing a CPU-intensive task), libuv holds incoming requests in a queue. Once the server is ready to process new requests, libuv forwards them to Node.js. This mechanism ensures efficient handling of I/O operations and concurrency in Node.js, but it doesn't prevent the blocking of the event loop by synchronous, CPU-intensive tasks.

Solution: spin up a new process (child process) to delegate

Nodejs handbook - https://flaviocopes.com/books-dist/node-handbook.pdf

Let us see some other important modules before arriving at the soln

1. OS module
    a. Create a new file os.js

```
const os = require('os');


console.log("arch",os.arch());
console.log("cpus",os.cpus());
console.log("freemem",os.freemem());
console.log("platform",os.platform());
console.log("release",os.release());
```

    b. os.arch(): Shows the architecture of the CPU, here it's arm64.
    c. os.cpus(): Lists details about each CPU core. The output shows an Apple M1 processor with 8 cores, displaying each core's model, speed in MHz, and time spent in different states (user, system, idle).

d. os.freemem(): Indicates the amount of free system memory in bytes. In this case, it's 137920512 bytes.
e. os.platform(): Displays the operating system platform, which is darwin (macOS).
f. os.release(): Shows the OS release version, here it's 23.0.0.

2. Network interface
   a.
```
console.log(os.networkInterfaces())
```
   b.
   c. provides detailed information about the network interfaces (like Ethernet, Wi-Fi) on the machine.
3. Path module
   a. Windows typically uses backslashes (\) for file paths, while macOS and Linux use forward slashes (/).
   b. This difference can lead to compatibility issues in Node.js applications running across different operating systems.
   c. The path module in Node.js helps to handle these differences. It provides a consistent API to work with file paths, allowing you to build paths that are correctly formatted for the operating system on which your Node.js application is running.
   d. Using path.join(), path.resolve(), and other methods from the path module ensures your file paths are correct regardless of the operating system.
   e. Create a new file called path.js

```
const path = require('path');
```

```
console.log(__dirname);
```

f. Get the base name - parent folder

```
const path = require('path');

console.log(__dirname);

const base = path.basename(__dirname);
console.log(base);
```

g. Create path independent of platforms

```
const path = require('path');

console.log(__dirname);

const base = path.basename(__dirname);
console.log(base);
const newPath =
path.join(__dirname,'public','abc','file.txt');
console.log(newPath);
```

4. Fs module

a. Create fs.js

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

// create a file
fs.writeFile( 'file.txt', 'hello world', (err) => {
   if(err) throw err;
   console.log("data written to file");
```

```
})
// add content to the file
fs.appendFile('file.txt', 'some more text', (err) => {
   if(err) throw err;
   console.log("data appended to file");
})
// read the file
fs.readFile('file.txt', (err, data) => {
   if(err) throw err;
   console.log(data.toString());
})
// create a directory
fs.mkdir('newDir', (err) => {
   if(err) throw err;
   console.log("Directory created");
})
// create another directory
fs.mkdir(path.join(__dirname,'newDir2'), (err) => {
   if(err) throw err;
   console.log("Directory created");
})
```

a. Copy file from models folder to current directory

```
// copy files from modules folder in the current
folder
const copyFrom =
path.join(__dirname,'../','models','bookingModel.
js');
const destPath =
path.join(__dirname,'bookingModelCopy.js');
```

```
fs.copyFile(copyFrom, destPath, (err) => {
   if(err) throw err;
   console.log("File copied");
})
```

Streams

1. In a Node.js application, how can you efficiently handle large files, like logs or media, without consuming excessive memory or blocking other operations?"
2. in Node.js, streams allow data to be processed in smaller chunks, one piece at a time, without needing to load the entire data into memory. This is like processing a large file bit by bit, instead of loading the whole file at once, which is memory-efficient and keeps the application responsive
3. Http request / response, crypto and some methods on fs module are internally stream enabled
4. Streams make use of of zlib
    a. zlib is often used behind the scenes in libraries and frameworks to manage data efficiently, particularly in network communications and file management. Browsers also use this
    b. zlib in Node.js plays a crucial role in handling large data by providing data compression and decompression capabilities.

    c. zlib is used in various file formats that require compression. For example, it's used in PNG image files, ZIP files, and gzip files, among others.
    d. zlib integrates well with Node.js streams, allowing for the compression or decompression of data streams on the fly.
5. So the idea of streams is not only limited to files but extend to other aspects like http, encryption, etc.
6. 4 types of stream
    a. Readable - stream to read the data (fs.createReadStream) , http req object
    b. Writable - stream for writing the data (createWriteStream)
    c. Duplex - sockets
    d. Transformative - change form from one to another. output is computed from the input - zlib, crypto
7. EventEmitter class
    a. It provides a means for objects to emit custom events and to attach listeners to these events, facilitating asynchronous programming
    b. Many core modules in Node.js, like fs, http, and stream, use EventEmitter for handling events like data (when data is available for reading) or end (when the end of the stream is reached).
    c. It supports Node.js's non-blocking nature by allowing operations to emit events when completed.
    d. Streams in Node.js inherently use EventEmitter.

e. You have already used the server.on listener

```
server.on('listening', () => {
   console.log('Server is listening on port 3000');
 });
```

Code for reading stream

1. In the fs.js file, comment other code and add below

```
const filePath = path.join(__dirname,'big.file');
console.log(filePath);
const readableSteam = fs.createReadStream(filePath);
readableSteam.on('data', (chunk) => {
   console.log(`Received ${chunk.length} bytes of data.`)
   // console.log(chunk.toString());
})
readableSteam.on('end', () => {
   console.log("Finsihed reading file");
})
```

2. Notice that there is no memory spike in the activity monitor
3. Very similarly, if we want to be able to write in a stream like manner

```
const filePath = path.join(__dirname,'big.file');
console.log(filePath);
const readableSteam = fs.createReadStream(filePath);
const writableStream =
fs.createWriteStream('copyOfBig.file');
readableSteam.on('data', (chunk) => {
   console.log(`Received ${chunk.length} bytes of data.`)
   // console.log(chunk.toString());
```

```
    writableStream.write(chunk);
})
readableSteam.on('end', () => {
    writableStream.end();
    console.log("Finsihed reading and writing the file");
})
```

4. Pipe
    a. pipe() function is a method on Readable streams and is used to connect a readable stream to a writable stream. It automatically handles the data transfer from the readable stream to the writable stream.
    b. To simplify your code using the pipe method, you can replace the manual read and write operations with a single pipe() call.

```
const filePath = path.join(__dirname,'big.file');
console.log(filePath);
const readableSteam = fs.createReadStream(filePath);
const writableStream =
fs.createWriteStream('anotherCopyOfBig.file');
// readableSteam.on('data', (chunk) => {
//     console.log(`Received ${chunk.length} bytes of
data.`)
//     // console.log(chunk.toString());
//     writableStream.write(chunk);
// })
// readableSteam.on('end', () => {
//     writableStream.end();
```

```
//     console.log("Finsihed reading and writing the
file");
// })
readableSteam.pipe(writableStream);

readableSteam.on('error', (err) => {
   console.log("error while reading",err);
})
writableStream.on('error', (err) => {
   console.log("error while writing",err);
})
```

Fixing Problem statement 1

1. How will we fix the first problem statement of a client requesting for a large file
2. We create a read stream and then write the stream to response

```
server.on('request', (req, res) => {
   // fs.readFile('./big.file', (err, data) => {
   //     if(err) throw err;
   //     res.end(data);
   // })
   const src = fs.createReadStream('./big.file');
   src.pipe(res);
})
```

3. How is this possible
    a. Because res is a writable stream