Short Answers

1. What is Node.js?

Node.js is an open-source, single-threaded, backend JavaScript runtime environment that runs on the V8 engine. In other words, it allows JavaScript, originally designed as a frontend scripting language, to be used for backend development.

2. What are some differences between Node.js and JavaScript?

JavaScript is a programming language, whereas Node.js is a JavaScript runtime environment. JavaScript was originally designed as a frontend scripting language, but Node.js enables its use for backend server development. JavaScript runs in browsers using browser-specific engines, such as SpiderMonkey in Mozilla Firefox, Chakra in Microsoft Edge (before December 2024), or V8 in Google Chrome. Node.js also runs on the V8 engine, which powers Google Chrome, but Node.js is not built into web browsers. JavaScript does not have its own event loop but depends on its environment: when JavaScript is run in the browser, it depends on the browser's engine and runtime environment; when JavaScript is run in Node.js, it inherits Node.js's runtime environment. Node.js has a non-blocking I/O event loop, which means it can handle asynchronous tasks such as API calls without blocking the main thread. JavaScript uses ES module importing and exporting syntax, while Node.js uses CommonJS (`require()`) by default (although it also supports ES6 modules). Additionally, JavaScript has browser-specific APIs such as `DOM`, `Window`, and `fetch`, while Node.js has system-level APIs such as `fs`, `HTTP`, and `os`. This is why JavaScript does not have access to the file system (for security reasons), while Node.js does, through its built-in `fs` module.

3. What is npm?

`npm` stands for Node Package Manager, which is a package manager for JavaScript maintained by npm, Inc., a subsidiary of GitHub. It allows you to install third-party libraries such as React, Redux, Lodash, Express, Tailwind CSS, `dotenv`, `nodemon`, etc. It consists of a command line client, also called `npm`, and an online database of public and paid-for private packages, called the npm registery. The registry is accessed via the client, and the available packages can be browsed and searched via the npm website. The package manager and the registry are managed by npm, Inc. We use `npm install -g npm` to install `npm` globally, and `npm -v` to check its version. We use `npm init -y` to create a `package.json` file, and then run `npm install` or `npm i` to install general dependencies, or `npm install <dependency>` to install specific dependencies, such as `npm install dotenv` for the `dotenv` dependency. And we usually run a local server with `npm start` or `npm run dev`.

4. What is CommonJS? How does it differ from ES Modules? How do we import files into other files in Node.js?

CommonJS is the standard module system built into Node.js, and it is not supported in browsers. It defines a standard for structuring and managing code in reusable modules, allowing developers to share and import functionality across different files. Even though CommonJS is a module system, when we use it in the browser via the `<script>` tag in HTML, it is treated as regular `text/javascript` (whereas when we import or export modules using ES6 syntax, the type in the `<script>` tag would be `module`).

To export files, can use the following code that will be placed at the bottom of the file:

```
module.exports = name;
module.exports = {name, name, name};
```

To import files, we can use the following code that doesn't necessarily have to be at the top of the file, but usually is for best practices:

```
// no need to add the .js extension for JavaScript files, but we do need
to include the extension for other file types
const filename = require("./file/path/filename");
```

Finally, if you have a `.js` file that uses ES6 syntax for importing and exporting and in the `<script>` tag the type is set to `module`, but you also wish to use CommonJS syntax, you can change the file extension to `.cjs`. If you want Node.js to treat all `.js` files in the project folder as CommonJS modules, then go into the project folder's `package.json` file and add the line `{ "type": "commonjs" }` at the root level of the `JSON` object.

5. How can you make the server automatically restart when files are modified?

To make the server automatically restart, or to keep the server running when files are modified, we can use `nodemon`. `nodemon` is a convenient `npm` package that provides a CLI command to start a Node.js application and automatically restarts the server whenever it detects changes in the project files or directory. We use `npm i -D nodemon` to install `nodemon` as a local dependency, or `npm i -g nodemon` to install it globally. Then, we use `nodemon filename.js` to run the server.

6. What are some global objects in Node.js?

Global objects in Node.js include `global`, which is the `global` object itself (in a browser this would be `window`); `console` for methods like `console.log()` and `console.error()`; `require()` and `module` for importing and exporting modules; and `process`, which provides information and control over the current Node.js process. These global objects do not need to be specifically imported using `require()`.

The `process` object has several important properties, including `process.arch`, `process.argv`, `process.env`, `process.release`, `process.version`, `process.pid`, and more. `process` also provides a few key functions, including `process.cwd()`, `process.memoryUsage()`, `process.on(event, listener)`, `process.kill(pid, [signal])`, `process.setuid(uid)`, `process.setgid(gid)`, `process.umask(mask)`, `process.exit([code])`, and others.

7. Explain how the Node.js architecture is event-driven in terms of event emitters and other core modules.

Node.js architecture is based on an event-driven model, where certain objects, called "emitters," emit named events that trigger the execution of associated functions, known as "listeners". The `EventEmitter` class in Node.js is a fundamental part of this model. When an event is emitted by an `EventEmitter` object, all listeners attached to that specific event are executed *synchronously*. This design is particularly effective

for handling asynchronous I/O operations, where tasks such as reading files, making HTTP requests, or querying databases are carried out without blocking the execution of other tasks. Any values returned by the listeners are ignored and discarded.

For example, a `fs.ReadStream` emits an event when the file is opened, and a stream emits an event whenever data is available to be read, as shown in the following code:

```
const EventEmitter = require("events");
const fs = require("fs");

// create a readable stream for the file named "file-text"
const readStream = fs.createReadStream("file-text");

// add a listener for the data event
readStream.on("data", function(chunk) {
    console.log("Reading a chunk of file data:", chunk);
});

// add a listener for the end event
readStream.on("end", function() {
    console.log("Finished reading the file.");
});

// add a listener for the error event
readStream.on("error", function(error) {
    console.error("Error reading the file:", error);
});
```

Many of Node.js's core objects, such as HTTP requests, responses, and streams, implement the `EventEmitter` class to facilitate emitting and listening to events. Below is an example of using the `EventEmitter` class:

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

class MyEmitter extends EventEmitter {};
const myEmitter = new MyEmitter();

myEmitter.on("event", () => {  // listen for the event using on()
  console.log("An event has occurred!");
});

myEmitter.emit("event");  // emit the event
```

8. What are streams? In Node.js, what are the different kinds of streams?

A stream is an interface in Node.js for handling large amounts of data efficiently by processing it in smaller chunks instead of loading the entire data into memory at once. Streams are particularly useful for I/O operations such as reading files, handling HTTP requests and responses, or working with sockets. There are

four types of streams in Node.js: writable streams, which allow data to be written to them, such as `HTTP` client requests, `HTTP` server responses, and `fs` write streams; readable streams, which allow data to be read from them, such as `HTTP` client responses, `HTTP` server requests, and `fs` read streams; duplex streams, which are both readable and writable, such as `net.Socket`; and transform streams, which are a special kind of duplex stream that can modify or transform the data as it is being read or written, such as `zlib` for compression or `crypto` for encryption. Streams provide a powerful and memory-efficient way to work with data in Node.js applications.

9. What is the difference between fs module's readFile and createReadStream?

The `fs` module in Node.js provides different methods for file operations, including `fs.readFile` and `fs.createReadStream`, both of which are used to read data from a file. The main difference between the two is that `fs.readFile` reads the entire file into memory before processing it. This method is suitable for small files where loading the entire file into memory will not cause performance issues. It is simple to use but not memory-efficient for handling larger files. Here is an example of `fs.readFile`:

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

fs.readFile("example.txt", "utf8", (error, data) => {
  if (error) throw error;
  console.log(data); // the entire content of the file
});
```

With `fs.createReadStream`, the file is read in chunks and processed incrementally as it is being read. This approach is ideal for large files where memory efficiency is important and non-blocking behavior is required. However, it is slightly more complex to use because it relies on handling events. Here is an example of `fs.createReadStream`:

```
const fs = require('fs');
const readStream = fs.createReadStream("example.txt", "utf8");

readStream.on("data", (chunk) => {
  console.log("Received chunk: ", chunk); // processes file data in chunks
});

readStream.on("end", () => {
  console.log("Finished reading the file.");
});
```

10. What are the different timing or scheduling functions in Node.js?

Some timing or scheduling functions in Node.js include `setTimeout(cb, delay, [args])`, `clearTimeout(timeoutObj)`, `setInterval(cb, interval, [args])`, `clearInterval(intervalObj)`, `setImmediate(cb, [args])`, `clearImmediate(immediateObj)`, and `process.nextTick(cb)`. `process.nextTick(callback)` schedules a function to be executed

immediately after the current operation completes, before the event loop continues to the next phase. This behavior is similar to how microtasks (promises) work, as both have higher priority than other tasks like `setTimeout` or `setImmediate`.

11. How does the event loop work in Node.js?

The event loop in Node.js is slightly different from the event loop in JavaScript in the browser, but there is still a callstack and a queue. The queue is still split into the macrotask queue and the microtask queue. These queues determine the sequence in which different types of callbacks are executed. The event loop alternates between processing microtasks and macrotasks. First, it executes all the microtasks, which include things like `process.nextTick()` and resolved promises. Once all the microtasks are done, the event loop processes one macrotask, such as a callback from `setTimeout()`, I/O events, or `setImmediate()`. After completing that macrotask, the event loop checks if there are any remaining microtasks and executes them before moving on to the next macrotask. This cycle continues, ensuring that microtasks are always handled before macrotasks, which helps the system stay responsive and efficient.

12. How do you manage multiple node versions?

We can use `nvm` (Node Version Manager) to manage multiple Node.js versions. It is a standalone package that allows us to install and switch between different Node.js versions (does not support Windows). After installing `nvm`, we can use several commands. The `nvm version` command checks the currently installed `nvm` version, while `nvm list node` shows the currently active Node.js version. To view all installed Node.js versions, including aliases like `node`, `stable`, and `lts`, we use `nvm list` (or `nvm ls`). To temporarily switch to a specific Node.js version for the current terminal session, we use `nvm use <version>`. Finally, the `nvm alias default <version>` command sets a specified Node.js version as the default for all future terminal sessions.