

Class 15 - Mern Interview - 2

Agenda

1. child process
2. Event and Event emitter
3. Http server
4. Event loop and architecture

Class starts

- i.
2. Problem statement 2
 - a. Node.js operates on a single-threaded, non-blocking, asynchronous event-driven architecture. This means that it can handle multiple requests concurrently without waiting for each one to complete before starting the next
 - b. In our backend code, when a request hits the /fib endpoint, the calculateFibonacci function is executed. This is a recursive function to calculate the Fibonacci number, which is CPU-intensive and blocks the event loop when dealing with large numbers. Despite Node's asynchronous nature, CPU-bound tasks like this can still block the event loop, leading to delays in processing other incoming requests.
3.
 - a. Consider C1, C2, C3, and C4 represent different clients.
 - b. Operating System Queue: When requests reach the server's operating system, they are placed in a queue. This is a

standard practice to manage incoming network requests. The operating system is responsible for managing these requests before they are handed off to the Node.js process.

- c. There is a handler function which processes the request and sends the response
- d. If the server is busy processing a request for client C1 (especially a CPU-intensive task like the Fibonacci calculation in our example), will it be able to process subsequent requests?
- e. No, This is because Node.js is single-threaded, and CPU-bound tasks can block the event loop.

Child Process

1. Child processes in Node.js allow you to perform operations in separate processes, which can be useful for CPU-intensive tasks or when you need to interact with the system at a lower level
4. With a child process, we can do 4 types of things
 - a. Exec -> can run any shell command
 - i. it runs a command for you and then gives you the entire output (output) at once, after the task is done
 - ii. When the exec method is called, it spawns a new shell and runs the command within that shell.
 - iii. The exec method is typically used for short-lived processes that require a shell environment,
 - iv.

V. `const { exec } = require('child_process');`

```
exec('ls -lh', (error, stdout, stderr) => {  
  if (error) {  
    console.error(`exec error: ${error}`);  
    return;  
  }  
  console.log(`stdout: ${stdout}`);  
  console.error(`stderr: ${stderr}`);  
});
```

vi. We should see list of contents in the directory with file permissions, owner, links to the file, memory, timestamp etc

b. `execFile` -> run any compiled file

- i. Similar to `exec`, but specifically for executing an executable file directly (as opposed to a command).
- ii. Create a script file `script.sh`
- iii. Enter the content below

```
#!/bin/bash  
echo "Script called with arguments: $1 $2"
```

iv. In the `childProcess.js` file, add the code below

```
const { execFile } = require('child_process');  
  
// The path to the script file  
const scriptPath = './script.sh'; // Use  
'script.bat' for Windows
```

```
// Arguments to pass to the script
const args = ['arg1', 'arg2'];

// Executing the script with arguments
execFile(scriptPath, args, (error, stdout,
stderr) => {
    if (error) {
        console.error(`Execution error:
${error}`);
        return;
    }
    console.log(`stdout: ${stdout}`);
    console.error(`stderr: ${stderr}`);
});
```

- v. Make sure to give script execute permission
- vi. `chmod +x script.sh`
- c. Spawn -> generally used to run different programs,
 - i. The spawn method in Node.js is used to spawn a new process and stream the output and error streams of that process.
 - ii. When the spawn method is called, it creates a new child process and streams the output and error streams of that process back to the parent process.
 - iii. The spawn method is typically used for long-lived processes that generate a large amount of output, such as a log collection process.

d. Fork ->

- i. The fork method in Node.js is used to create a new Node.js process. When the fork method is called, it creates a new child process that is a copy of the parent process, including all of the parent's memory and runtime
- ii. The fork method also allows for inter-process communication (IPC) between the parent and child processes
- iii. Fork creates a child process that is a copy of the parent process, while spawn creates a new process from scratch.

5. Fork vs spawn

- a. We Use fork when you need to create a new Node.js process that shares some or all of the parent process's memory and runtime environment, and when you need to communicate between the parent and child processes using IPC. eg running a cpu intensive task
- b. We Use spawn when you need to spawn a new process and stream its output and error streams back to the parent process. Basically just get the work done and report its status

Code

1. Create a new file childProcess.js

2. We will try to open a different application using spawn method
3. Get the executable path of the chrome from chrome://version
4. Incognito flag and others can be searched with chromium flags

```
const cp = require('child_process');

cp.spawn("/Applications/Google
Chrome.app/Contents/MacOS/Google Chrome",
["https://www.youtube.com/", "--incognito"])
```

5. Now lets use fork to take care of the fibonacci function
6. Create a file fiboWorker.js and move the code

```
function calculateFibonacci(number) {
  if (number <= 1) {
    return number;
  }
  return calculateFibonacci(number - 1) +
calculateFibonacci(number - 2);
}
```

6. Adding an event listener to trigger this function

```
function calculateFibonacci(number) {
  if (number <= 1) {
    return number;
  }
  return calculateFibonacci(number - 1) +
calculateFibonacci(number - 2);
}

// invoking this on process.on and event name
process.on('message', ({number}) => {
```

```
    const result = calculateFibonacci(number);  
    process.send(result);  
  })
```

7. Updating request handler

```
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);  
  // creating a child process  
  const fiboRes = fork(path.join(__dirname,  
'fibWorker.js'));  
  // sending data to the child process  
  fiboRes.send({ number: parseInt(number,10) });  
  // receiving data from the child process  
  fiboRes.on('message', (answer) => {  
    console.log("sending response for req",  
requestNumber);  
    res.status(200).json({  
      status: "success",  
      message: answer,  
      requestNumber  
    })  
    // kill the child process  
    fiboRes.kill();  
  })  
  // console.log(answer);
```

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

8. See the activity monitor for the different node processes that spin up and eventually die after job completion

Nice article wrapping everything that we have learnt -

<https://www.freecodecamp.org/news/node-js-child-processes-everything-you-need-to-know-e69498fe970a/>

Event Emitter class

1. In Node.js, an Event Emitter is a special type of class that allows objects to communicate with each other by emitting events and listening for them. Think of it as a system for sending and receiving signals or messages.
2. Emitting Events: An object created from the Event Emitter class can send out, or 'emit', events. These events are identified by a name (like 'click', 'error', 'data', etc.).
3. Listening for Events: Other parts of your program can 'listen' for these events. When an event is emitted, the corresponding

listener functions (also known as 'handlers') that were set up to react to that specific event are automatically called.

4. They are perfect for handling asynchronous operations in Node.js, where you might not know exactly when a certain action (like receiving data from a server) will happen.
5. We can define our own custom events
6. It provides the foundational mechanism for emitting named events and registering listener functions (event handlers) that are invoked when those events are emitted.

Code

1. Create a new file eventEmitter.js
2. We'll create a simple event handler

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

const myEmitter = new EventEmitter();

// listeners
myEmitter.on('myEvent', (...args) => {
  console.log("There is a new event!",args);
});

myEmitter.on('myEvent', (...args) => {
  console.log("another listener for the new event",args);
  console.log("-----")
});
```

```
// emit an event
myEmitter.emit('myEvent' );
myEmitter.emit('myEvent', 1,2);
myEmitter.emit('myEvent', [1, 2,3]);
```

3. Now we can selectively remove callbacks from our events

a. Let us create a named callback function

```
const secondCb = (...args) => {
  console.log("another listener for the new event",
args);
  console.log("-----");
};

// listeners
myEmitter.on("myEvent", (...args) => {
  console.log("There is a new event!", args);
});

myEmitter.on("myEvent", secondCb);
```

b. For the third event , let us remove the second callback

```
// emit an event
myEmitter.emit("myEvent");
myEmitter.emit("myEvent", 1, 2);
myEmitter.off('myEvent', secondCb);
myEmitter.emit("myEvent", [1, 2, 3]);
```

Http servers

1. Create a basic server

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

const server = http.createServer();

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

2. Let us add a listener to listen to the requests

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

const server = http.createServer();

server.on("request", (req, res) => {
  console.log("headers", req.headers, "url", req.url,
"method", req.method);
  console.log("request event");
  res.end("Hello World");
});

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

3. In console, you might see few things

- a. accept: '*/*': This header indicates that the client will accept any type of data in response.
- b. 'accept-encoding': 'gzip, deflate, br': This indicates the type of encoding (compression) the client can handle in the response. It can accept gzip, deflate, or Brotli compressed content.
- c. connection: 'keep-alive': This suggests that the client wants to keep the network connection open after the current transaction complete

4. Segregating based on http methods

```
server.on("request", (req, res) => {  
  console.log("headers", req.headers, "url", req.url,  
"method", req.method);  
  console.log("request event");  
  if (req.method === "GET") {  
    res.writeHead(200, {  
      "Content-Type": "text/plain",  
    });  
  } else if (req.method === "POST") {  
    res.writeHead(200, {  
      "Content-Type": "application/json",  
    });  
  }  
  res.end("Hello World");  
});
```

- a. MIME types are essential in the web context for determining how browsers process a URL. MIME types consist of a type and a subtype (like text/plain or

image/jpeg) and are used by web servers to inform browsers about the type of the content being served.

b. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

c. for example, if we are serving HTML files, we should set the header to text/html. For JSON data, use application/json

d. There is something called as MIME sniffing.

i. a web browser or server attempts to determine the content type of a document or file based on its content, rather than relying solely on the Content-Type header. This can be exploited in security attacks such as delivering malicious code. For example, an attacker could upload a specially crafted file with an ambiguous MIME type, which is then executed as a script by the browser, leading to cross-site scripting (XSS) vulnerabilities.

ii. This is why it's important to properly set Content-Type

5. Update the GET method, to write some response

a. Res.write is used to update the res stream and end is when the response is sent

```
if (req.method === "GET") {  
  res.writeHead(200, {  
    "Content-Type": "text/plain",  
  });  
  res.write("Hello World");  
  res.write("Hello World Again");  
  res.end();  
}
```

```
}
```

b. In the postman verify the response

6. Update the post method to send back a response

```
res.end({  
  name: "John",  
})
```

7. Hit a post request from postman, this should give an error

8. The chunk is expected to be of type string but we are sending object

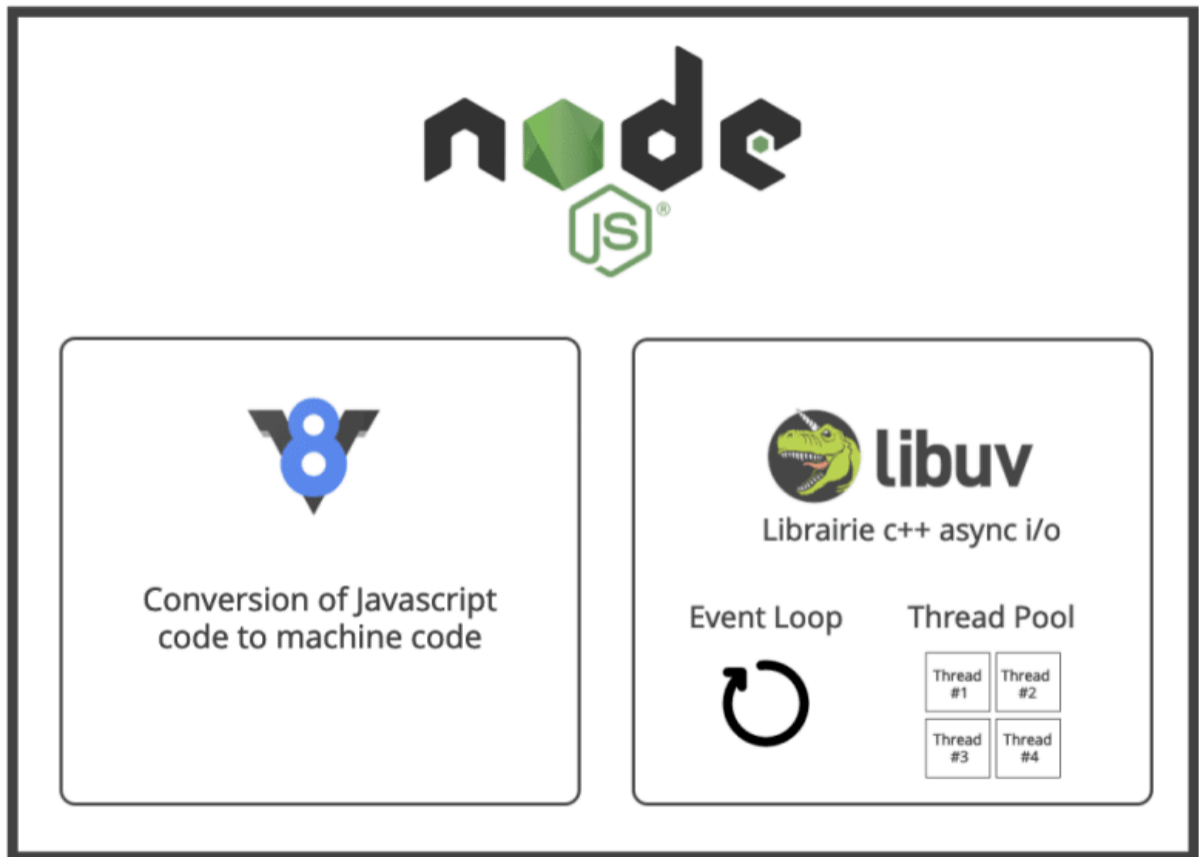
9. What we need to do is stringify before sending

```
res.end(JSON.stringify({name: "John"}));
```

10. Currently we are doing conditional response handling based on methods, if you were creating a full fledged server then first you will be checking for routes and then for each route, you will have different methods

11. Now you should appreciate how easy express makes our life

Node Architecture



- 1.
2. The two main parts of node js is the v8 engine and the libuv
3. libuv provides Node.js with event-driven, asynchronous I/O capabilities. It implements the event loop and a thread pool, which allows Node.js to handle many operations concurrently, such as file and network operations. This is key to Node.js's non-blocking behavior.
4. The Thread Pool can run tasks in parallel and therefore takes care of more cumbersome tasks such as access to the file system and very demanding processes such as for example video conversions or cryptography.
5. <https://libuv.org/>

6. In the documentation, you will find multiple points
 - a. Full-featured event loop backed by epoll, kqueue, IOCP, event ports.
 - i. epoll, kqueue, IOCP, and event ports are all mechanisms provided by different operating systems to handle asynchronous I/O operations.
 - ii. Epoll used by linux
 - iii. loep - microsoft
 - iv. Kqueue is found in BSD systems, including macOS
 - v. Libuv provides a robust event loop mechanism that is compatible with various system-level asynchronous I/O operations (reading / writing files, database operations , etc)

When can Nodejs said to be multithreaded or when does libuv offer multithreading support to node

1. <https://nodejs.org/en/guides/dont-block-the-event-loop/>
- 2.

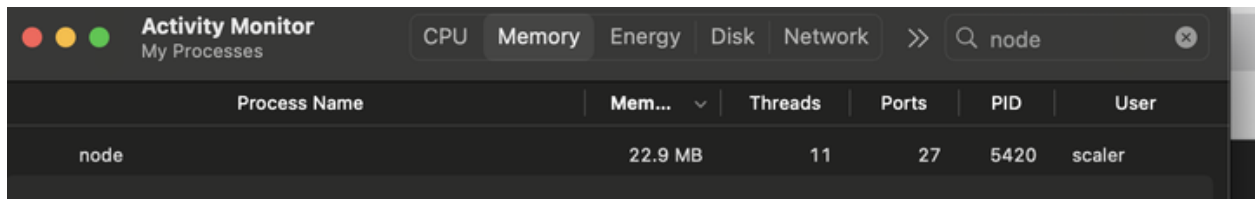
1. I/O-intensive

1. **DNS:** `dns.lookup()` , `dns.lookupService()` .
2. **File System:** All file system APIs except `fs.FSWatcher()` and those that are explicitly synchronous use libuv's threadpool.

2. CPU-intensive

1. **Crypto:** `crypto.pbkdf2()` , `crypto.scrypt()` , `crypto.randomBytes()` , `crypto.randomFill()` , `crypto.generateKeyPair()` .
2. **Zlib:** All zlib APIs except those that are explicitly synchronous use libuv's threadpool.

1. While Node.js's main execution thread is single-threaded, it offloads certain I/O-intensive and CPU-intensive tasks to a thread pool managed by Libuv.
2. This allows operations like DNS lookups, file system processing, cryptographic computations, and zlib compression to be processed in parallel, without blocking the main event loop.
3. for the mentioned tasks, Node.js uses its worker pool to behave in a multi-threaded manner

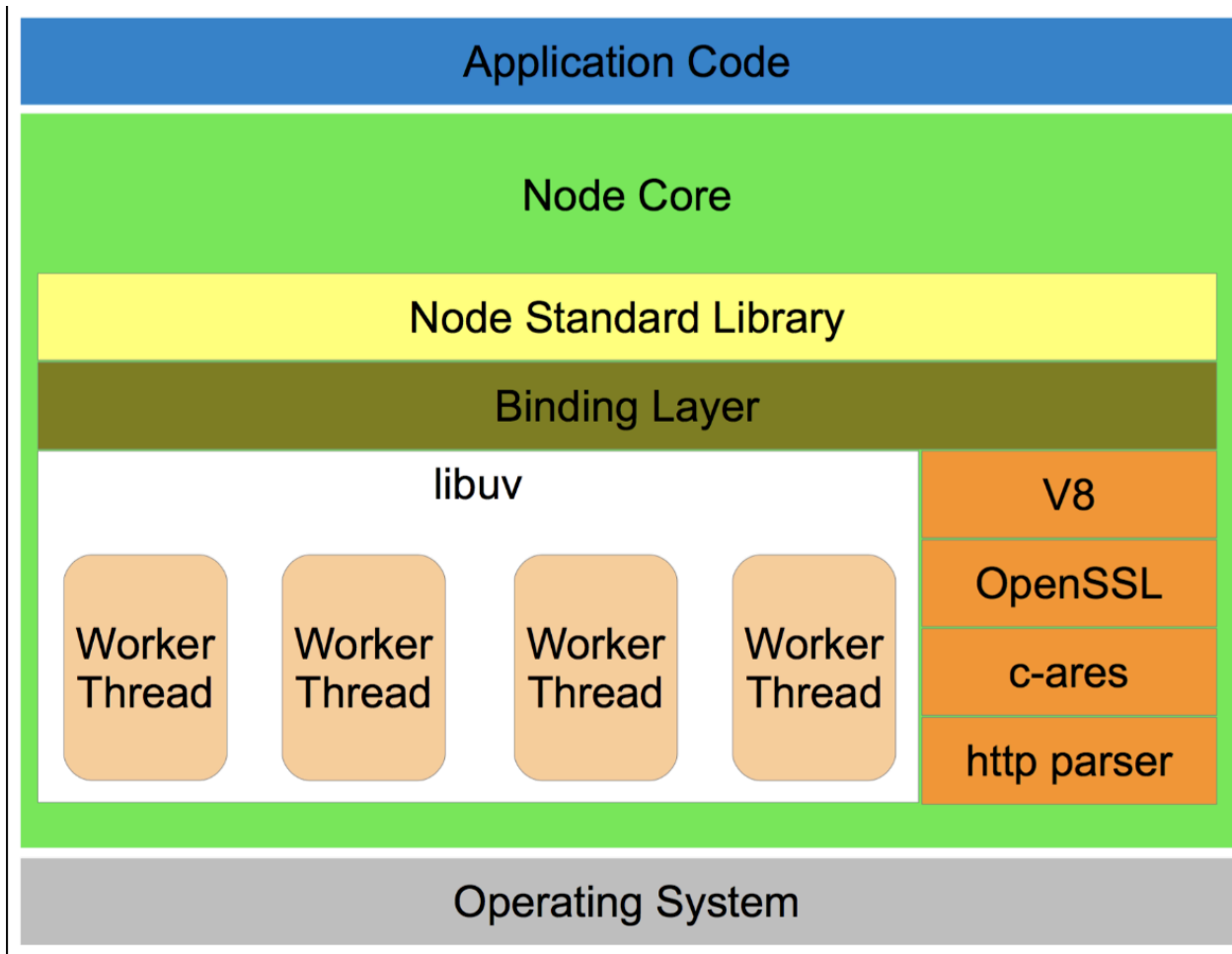


The screenshot shows the macOS Activity Monitor window with the 'Memory' tab selected. A search bar at the top right contains the text 'node'. Below the search bar, a table lists the details of the 'node' process. The table has columns for Process Name, Memory (Mem...), Threads, Ports, PID, and User. The 'node' process is listed with 22.9 MB of memory, 11 threads, port 27, PID 5420, and user 'scaler'.

Process Name	Mem...	Threads	Ports	PID	User
node	22.9 MB	11	27	5420	scaler

4. The threads that we see in the activity monitor are part of Node.js's internal thread pool, which it uses to perform various tasks that are offloaded from the main thread

Architecture discussion

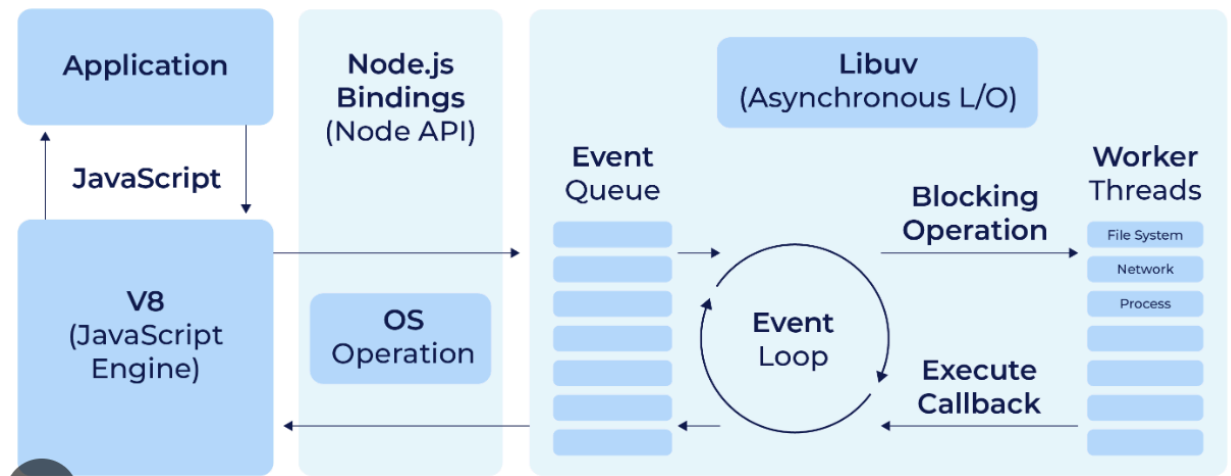


1. Application code - has the javascript code
2. The core of Node.js includes the JavaScript runtime and the basic modules required to make Node.js function as a server-side platform.
3. Binding layer is the interface or the bridge between your JS and the internal C / C++ code
 - a. This layer acts as a bridge between the JavaScript code running on the V8 engine and the lower-level operations handled by Node's C/C++ components. It translates

JavaScript calls into actions that can be executed at the system level.

4. Node Standard library are the modules - fs, os, http
5. Libuv is in C
6. V8 is in c++
7. OpenSSL: This is a software library used for secure communication over networks through the use of cryptography protocols like SSL and TLS.
8. c-ares: This is a C library that performs DNS requests
9. http-parser: A lightweight C library used by Node.js to parse HTTP messages

Node.js Architecture



Order of execution:

1. Application Code Execution: You write JavaScript code that is executed by the V8 engine.

2. V8 Engine: It compiles JavaScript into machine code.
3. Node Bindings: These serve as a bridge between your JavaScript code and Node.js's C/C++ core functionalities.
4. Libuv & Other C/C++ Modules: If the JavaScript code requires it, Node.js will use Libuv for handling I/O tasks or other C/C++ libraries for things like cryptographic functions.
5. Callback to V8: Once the I/O or other operations are complete, the results are passed back to V8, often through callbacks.
6. Application Code: Finally, the callback or result is returned to your application code for further processing or response to the client.

Event LOOP

1. <https://nodejs.org/en/guides/event-loop-timers-and-nexttick/>
2. Initialization: When a Node.js process starts, it initializes the event loop.
3. Phases: The event loop runs through several phases, each with its own queue of callbacks to execute.
4. Poll Phase: The event loop enters the poll phase where it will execute I/O callbacks and awaits new events.
5. Check Phase: After the poll phase, it checks for `setImmediate()` callbacks and executes them.

6. Close Callbacks Phase: Any close callbacks, like `socket.on('close', ...)` are executed.
7. Timers Phase: The event loop checks for any `setTimeout()` or `setInterval()` callbacks ready to be called.
8. Process Next Tick: Before moving to the next phase, Node.js processes `process.nextTick()` callbacks, which are executed as soon as the current operation completes.
9. Microtasks: After each phase, microtasks (such as resolved Promise callbacks) are processed.
10. Repeat: This loop continues until there are no more callbacks to process, then Node.js can exit.

Summarising

1. Node.js uses an event loop for non-blocking I/O operations by offloading operations to the system kernel whenever possible
2. The event loop operates in phases: timers, pending callbacks, idle/prepare, poll, check, and close callbacks.
3. Timers (`setTimeout` and `setInterval`) schedule callbacks to run after a minimum threshold.
4. `process.nextTick()` schedules a callback to execute immediately after the current operation completes.
5. The poll phase is critical as it decides when to execute I/O tasks and their callbacks.
6. `setImmediate()` is used for callbacks that should run after the poll phase of the event loop.

7. Using `process.nextTick()` allows handling operations right after the current script finishes but before returning to the event loop.
8. it's important to understand that `process.nextTick()` and `setImmediate()` serve different purposes, with the former executing callbacks immediately after the current operation and the latter after the current event loop phase.

Code

1. Create a new `eventLoop.js`

```
console.log('Start');

process.nextTick(() => {
  console.log('Next Tick');
});

setImmediate(() => {
  console.log('Set Immediate');
});

console.log('End');
```

2. The `process.nextTick` callback runs right after the current script ('Start' and 'End' log statements) finishes, even before `setImmediate`, which waits until the next cycle of the event loop.