# Event Loop in NodeJS

The event loop in NodeJS is a mechanism that allows asynchronous tasks to be handled efficiently without blocking the execution of other operations.

1. Executes JavaScript synchronously first and then processes asynchronous operations.

2. Delegates heavy tasks like I/O operations, timers, and network requests to the libuv library.

3. Ensues smooth execution of the multiple operations by queuing and scheduling callbacks efficiently.

NodeJS operates in a single thread.

The Event loop is essential in Node.js because it allows non-blocking,asynchronous operations to be handled efficiently,even though Node.js operates on a single thread.

1. Enables non-blocking execution despite Nodes.js being single-threaded.

2. Helps handle I/O tasks efficiently.

3. Makes NodeJS suitable for scalable applications like web servers.

console.log("This is the first statement");

setTimeout(function() {

console.log("This is the second statement");

}, 1000);

console.log("This is the third statement");

First statement is logged immediately.

setTimeout schedules the second statement to log after 1000 milliseconds.

Third statement is logged immediately after.

After 1000 milliseconds, the callback from setTimeout is executed, logging the Second statement.

# 

# How the Event Loop Works?

Synchronous code first and then moves to handle asynchronous tasks.

1. Initialization (When Node.js starts, it loads the script, executes synchronous code, and registers any asynchronous tasks)

2. Execution of Input Script (The call stack executes synchronous code first.)

3. Handling Asynchronous operations with libuv (Node.js uses a special C library called libuv to handle asynchronous operations. This library manages a thread pool that offloads heavy tasks (like file I/O, database operations, or network requests) that would otherwise block the event loop.)

4. Callback Execution (Once the thread pool completes its tasks, it sends callbacks to the event queue. Execute only When No synchronous code is currently executing)

5. Event Loop Phases (It checks for events, handles asynchronous callbacks, and executes tasks in the correct order.)

6. Callback Execution from Event Queue

### Phases of the Event Loop

#### 1. Timers Phase : This phase processes timers that have been set using setTimeout() and setInterval()

console.log('Start');

setTimeout(() => {

console.log('Timeout callback');

}, 2000);

console.log('End');

#### 2. Pending Callbacks: This phase executes I/O related callbacks that were deferred from the previous loop cycle.

console.log('Start');

setImmediate(() => {

console.log('Immediate callback');

});

console.log('End');

#### 3. Idle, Prepare (Internal use only): This phase is used internally by [Node.js](http://node.js) for background tasks.

#### 4. Poll Phase (Main Phase): The Poll phase executes most of the tasks like- I/O, file reading, HTTP requests and much more.

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

#### const readStream = fs.createReadStream('./file.txt');

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

#### 

#### readStream.on('data', (chunk) => {

#### console.log(chunk.toString());

#### });

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

#### 5. Check Phase : This phase processes any setImmediate() callbacks that have been added to the message queue.

console.log('Start');

setImmediate(() => {

console.log('Immediate callback');

});

console.log('End');

#### 6. Close Callbacks Phase: This phase executes for closed connections like sockets, streams, and event emitters.

const net = require('net');

const server = net.createServer((socket) => {

socket.on('close', () => {

console.log('Socket closed');

});

});

server.listen(8000);

## process.nextTick() and Promises in the Event Loop

Apart from these phases there is also process.nextTick() and promise callback which has the highest priority in the event loop.

* process.nextTick() callbacks are always executed before the event loop moves to the next phase.
* Resolved Promise callbacks are processed immediately after process.nextTick().

setImmediate(() => {

console.log("setImmediate is called");

});

Promise.resolve("Promise is resolved").then(console.log);

setTimeout(() => {

console.log("Time function is called");

}, 0);

process.nextTick(() => {

console.log("Process.nextTick");

});

# What is Promise Chaining?

Promise Chaining in [Node.js](http://node.js) allows you to execute multiple asynchronous operations in a sequence, where each operation starts after the previous one completes.

1. Sequential Execution: Makes sure asynchronous tasks run in a specific order,each task waiting for the previous one’s results.
2. Error Handling: Provides a clean way to handle errors in asynchronous operations, making sure any problems are caught and dealt with.

promise

.then(value => {

return newValue;

})

.then(newValue => {

return anotherValue;

})

.catch(error => {

});

# How Promise Chaining Works?

1. Each .then() receives the results from the previous step.you can use that result and then pass something new along to the next step.
2. Returning a value from .then() automatically creates a new promise that carries that value. This keeps the chain going. If you return a promise from .then(), the chain waits for that promise to finish before moving on.
3. The .catch() at the end is like a safety net. It catches any error that might have occurred in any of the steps in the chain.

const myPromise = new Promise((resolve, reject) => {

setTimeout(() => { resolve(10); }, 500);

});

myPromise

.then(value => {

console.log("First then:", value);

return value \* 2;

})

.then(newValue => {

console.log("Second then:", newValue);

return newValue + 5;

})

.then(finalValue => {

console.log("Final value:", finalValue);

})

.catch(error => { console.error("Error:", error); });

## **Promise Chaining vs. Async/Await**

While Promise chaining improves readability compared to callbacks, async/await offers an even more synchronous-like approach.

Async function fetchDetails(){

try{

Const user=await fetchData(1);

console.log(“User fetched:”[user.name](http://user.name));

Const orders=await fetchOrders(user);

console.log(“Order fetched:”,orders);

Const processedOrder=await processOrder(orders[0]);

console.log(“Final Results:”,processedOrder);

} catch (error){

console.log(“Error:”,error);

}

}

fetchDetail();

# API Making in NodeJS

const http=require(“http”);

const url=require(“url”);

Let users=[

{id:1,name:”Alice”},

{id:2,name:”Bob”},

];

Const server=http.createServer((req,res)=>{

const parsedUrl=url.parse(req.url,true);

Const {pathname}=parsedUrl;

res.setHeader(‘Content-Type’,’application/json’);

if(req.method===’GET’ && pathname===’/users’){

res.writeHead(200);

res.send(JSON.stringify(users));

}

else if(req.method=== ‘POST’ && pathname === ‘/users’){

Let body=’’;

[req.on](http://req.on)(‘data’,chunk=>(body+=chunk));

[req.on](http://req.on)(‘end’,()=>{

const newUser=JSON.parse(body);

[newUser.id](http://newuser.id)=users.length+1;

users.push(newUser);

res.end(JSON.stringify(newUser));

});

}

Else{

res.writeHead(404);

res.end(JSON.stringify({message:”Not found”}));

}

});

server.listen(3000,()=>{

console.log(‘Server running at port 3000’);

});

# API Using [Express.JS](http://express.js)

Const express=require(‘express’);

Const app=express();

app.use((req,res,next)=>{

console.log(“Middleware Called!”);

next();

});

app.get(‘/’,(req,res)=>{

res.send(“Hello World!”);

});

app.listen(3000);

# Middleware in Express

Middleware in Express refers to functions that process requests before reaching the route handlers. These functions can modify the request and response objects, end the request-response cycle, or call the next middleware function. Middleware functions are executed in the order they are defined. They can perform tasks like authentication, logging or error handling. Middleware helps separate concerns and manage complex routes efficiently.

# Application-level Middleware

# Application-level middleware is bound to the entire Express application using app.use() . It executes for all routes in the application, regardless of the specific path or HTTP method.

1. Router -level Middleware

Router-level Middleware is applied to a specific router instance using router.use(). It only applies to routes defined within that particular router, making it perfect for modular applications where middleware is only relevant to specific groups of routes.

1. Error-handling Middleware

Error-handling middleware is a special type of middleware used to catch and respond to errors during the request-response cycle.It is defined with four parameters: err, req, res, next.

1. Built-in Middleware

Express provides built-in middleware to help with common tasks, like serving static files or parsing data.

1. Third-party Middleware

Third party middleware is developed by external developers and packaged as npm modules.

# NodeJs EventEmitter

The EventEmitter class in NodeJS is a core module that provides a way to handle asynchronous events. It allows objects to emit events and other objects to listen and respond to those events.

Event Emitter in Node

NodeJS uses an events module to create and handle custom events. The EventEmitter class can be used to create and handle a custom events module.

### **Importing EventEmitter**

EventEmitter is a class

const express=require(“express);

const EventEmitter=require(“events”);

Const app=express()

app.get(“/”,(req,res)=>{

res.send(“API called”);

});

app.listen(8000);

### Basic Structure of Event Emitter

Const EventEmitter=require(‘events’);

Const emitter=new EventEmitter();

[emitter.on](http://emitter.on)(‘greet’,()=>{

console.log(“Hello from EventEmiiter!”);

});

emitter.emit(‘greet’);

### Example with Arguments

emitter.on('message', (name) => {

console.log(`Hello, ${name}!`);

});

emitter.emit('message', 'Alice'); // Hello, Alice!

## MVC Structure Of NodeJS

MVC stands for:

1. M-Model (Handles data & database logic)
2. V-view (Handles what the user sees (UI))
3. C-Controller (Handles logic between model and view)
4. User makes a request (e.g., Click a button)
5. Controller handles the request
6. Controller talks to the Model to get or update data
7. Model sends data back
8. Controller sends data to View
9. View shows the data to the user