# Hooks

In Node.js, **hooks** refer to a system for intercepting and reacting to various phases of an operation or lifecycle. Node.js provides hooks in different contexts, such as **Async Hooks**, **Lifecycle Hooks**, and **HTTP Hooks**, among others. Each serves a distinct purpose in monitoring or extending the behavior of a system during specific events or phases.

**Types of Hooks in Node.js**

**1. Async Hooks**

**Async Hooks** provide an API to track asynchronous resources throughout their lifecycle. It helps developers monitor and track the execution context of asynchronous operations (such as timers, file system I/O, and network requests) by defining hooks at different phases of an async task.

**Lifecycle Phases (Events):**

* **init**: Called when a new async resource is created.
* **before**: Called just before an async callback is about to execute.
* **after**: Called after the async callback finishes execution.
* **destroy**: Called when an async resource is destroyed.
* **promiseResolve**: Specific to promises, called when a promise is resolved.

Example:

js

Copy code

const async\_hooks = require('async\_hooks');

const hook = async\_hooks.createHook({

init(asyncId, type, triggerAsyncId) {

console.log(`Init: ${asyncId}`);

},

before(asyncId) {

console.log(`Before: ${asyncId}`);

},

after(asyncId) {

console.log(`After: ${asyncId}`);

},

destroy(asyncId) {

console.log(`Destroy: ${asyncId}`);

}

});

hook.enable();

setTimeout(() => {

console.log('Async operation');

}, 1000);

**2. Lifecycle Hooks in HTTP Servers**

In an HTTP server (like Express.js), lifecycle hooks refer to middleware functions that are executed at various stages of the request-response cycle. These hooks can be used to modify requests, responses, or handle errors.

**Types of Hooks in Express.js:**

* **Application-level Middleware**: Hooks that apply to the entire app.
* **Router-level Middleware**: Hooks that apply to specific routes or groups of routes.
* **Error-handling Middleware**: Hooks that handle errors occurring during request processing.
* **Third-party Middleware**: Pre-built middleware for specific tasks (e.g., logging, parsing).

Example of an Express.js middleware hook:

js

Copy code

const express = require('express');

const app = express();

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

console.log('Middleware hook - Request received');

next();

});

app.get('/', (req, res) => {

res.send('Hello, world!');

});

app.listen(3000);

**3. Hooking into the Event Loop**

Node.js also provides the ability to hook into the event loop via the setImmediate, process.nextTick, and setTimeout functions. These functions can be used to schedule tasks at various stages of the event loop execution cycle.

* **process.nextTick()**: Defers the execution of a function until the next iteration of the event loop, before any I/O operations.
* **setImmediate()**: Executes a callback on the next iteration of the event loop, but after I/O events.
* **setTimeout()**: Defers the execution of a function by a specified time.

Example:

js

Copy code

process.nextTick(() => {

console.log('Executed in the next event loop iteration');

});

setImmediate(() => {

console.log('Executed after I/O events');

});

setTimeout(() => {

console.log('Executed after 100ms');

}, 100);

**4. Process Hooks**

Process hooks allow developers to react to specific process-level events, such as when the process exits, when an unhandled promise rejection occurs, or when an uncaught exception is thrown. These hooks are particularly useful for clean-up or logging in critical situations.

Common hooks:

* **process.on('exit')**: Triggered just before the Node.js process exits.
* **process.on('uncaughtException')**: Triggered when an exception goes unhandled.
* **process.on('unhandledRejection')**: Triggered when a Promise is rejected without a catch handler.

Example:

js

Copy code

process.on('exit', (code) => {

console.log(`Process is exiting with code: ${code}`);

});

process.on('uncaughtException', (err) => {

console.error('There was an uncaught error:', err);

process.exit(1);

});

// Simulate an uncaught exception

throw new Error('Test Error');

**Summary**

Hooks in Node.js offer powerful mechanisms for monitoring, modifying, and extending functionality across asynchronous operations, HTTP requests, and process-level events. Each type of hook serves specific needs in terms of lifecycle monitoring, context propagation, and error handling.

**Performance Hooks** in Node.js provide a mechanism to monitor and measure the performance of Node.js applications. The **Performance API** (part of the perf\_hooks module) allows developers to track and analyze timing information related to different parts of their code, such as event loop delays, execution time of specific functions, or third-party library performance.

**Key Concepts and Components**

1. **Performance Object**: The central object in the Performance API, responsible for recording high-resolution timestamps and measuring durations between marks.
   * Similar to the performance object in the browser's Web Performance API.
2. **High-Resolution Timestamps**: All timestamps are represented in milliseconds with microsecond precision, making this suitable for measuring even very short intervals.
3. **Marks and Measures**: These are custom time markers used to measure performance within an application.
   * **mark()**: Used to create a timestamp at a specific point in your code.
   * **measure()**: Measures the duration between two marks or from a mark to the current time.
4. **Event Loop Delay**: The performance API also provides utilities to monitor the event loop delay, which is useful in diagnosing performance bottlenecks.

**Example Use Cases**

**1. Basic Performance Measurement**

To measure how long a section of your code takes to execute:

js

Copy code

const { performance, PerformanceObserver } = require('perf\_hooks');

// Create performance marks

performance.mark('start');

// Simulate some code execution

setTimeout(() => {

performance.mark('end');

performance.measure('Execution Time', 'start', 'end');

// Access the measured results

const entries = performance.getEntriesByType('measure');

console.log(entries);

}, 1000);

In this example, the code:

* Marks the start of an operation.
* Marks the end after a setTimeout.
* Measures the time taken between the two marks and logs the result.

**2. Monitoring Event Loop Delay**

The monitorEventLoopDelay() function measures the delay in the event loop, providing insights into how smoothly the event loop is running. Event loop delays can indicate performance issues if operations block it.

js

Copy code

const { monitorEventLoopDelay } = require('perf\_hooks');

// Create an event loop monitor

const h = monitorEventLoopDelay({ resolution: 10 });

// Enable the monitor

h.enable();

setTimeout(() => {

h.disable();

console.log(`Event loop delay: ${h.mean.toFixed(2)} ms`);

}, 1000);

In this example, monitorEventLoopDelay() tracks how much time the event loop is being delayed, providing insights on bottlenecks.

**3. Using PerformanceObserver to Track Performance Events**

The PerformanceObserver class can be used to observe performance entry events in real-time.

js

Copy code

const { performance, PerformanceObserver } = require('perf\_hooks');

// Create an observer

const obs = new PerformanceObserver((list) => {

const entries = list.getEntries();

entries.forEach(entry => {

console.log(`${entry.name}: ${entry.duration}ms`);

});

});

// Start observing performance events

obs.observe({ entryTypes: ['measure'] });

// Performance measurements

performance.mark('start');

setTimeout(() => {

performance.mark('end');

performance.measure('Timeout duration', 'start', 'end');

}, 500);

In this code:

* The observer tracks any performance measurements.
* Every time a measurement is made (e.g., the Timeout duration), it is logged with its duration.

**4. Performance Timeline and Entry Types**

The Performance API supports several built-in entry types in addition to custom marks:

* **node**: Captures details about Node.js startup.
* **gc**: Tracks garbage collection cycles.
* **http2**: Measures performance related to HTTP/2 operations.
* **function**: Captures performance for specific JavaScript function executions.

For example, to track garbage collection events:

js

Copy code

const { PerformanceObserver } = require('perf\_hooks');

const obs = new PerformanceObserver((list) => {

const entries = list.getEntries();

entries.forEach(entry => {

console.log(`Garbage collection: ${entry.kind} took ${entry.duration}ms`);

});

});

obs.observe({ entryTypes: ['gc'] });

**Common Performance Hooks Functions**

1. **performance.now()**: Returns a high-resolution timestamp that represents the time elapsed since the Node.js process started. This is useful for measuring precise time durations.

Example:

js

Copy code

const start = performance.now();

// Some operation

const end = performance.now();

console.log(`Duration: ${end - start}ms`);

1. **performance.mark()**: Creates a performance entry that acts as a named time point in your application’s lifecycle. Useful for setting start and end points.
2. **performance.measure()**: Measures the duration between two performance marks or between a mark and the current time.
3. **performance.clearMarks()**: Clears any recorded marks. Useful when you want to reset marks between measurements.
4. **performance.clearMeasures()**: Clears recorded measures, cleaning up memory after recording performance data.

**Summary**

The Performance Hooks in Node.js allow developers to create and monitor performance metrics with high precision. This is useful for:

* Benchmarking code sections.
* Monitoring event loop delays.
* Tracking garbage collection cycles.
* Analyzing custom or built-in performance data for HTTP, HTTP/2, and Node.js core.

This API is especially valuable for optimizing performance-sensitive applications, particularly when dealing with large-scale or time-critical Node.js apps.

# WebCrypto API

**Differences between crypto and WebCrypto**

* **API Style**: WebCrypto is Promise-based, while crypto can be synchronous or callback-based.
* **Key Management**: WebCrypto restricts access to raw key material (for security reasons), while Node's crypto allows greater flexibility in managing keys.
* **Portability**: WebCrypto is designed to work in both browsers and server environments, making it ideal for web applications that need to share cryptographic logic between frontend and backend.
* **Functionality**: The crypto module in Node.js offers a broader range of algorithms and features, including file-based encryption and higher-level cryptographic tools, while WebCrypto is more focused on web-standard cryptography.

# Asynchronous context tracking

**Asynchronous context tracking** refers to the ability to maintain and track the execution context across asynchronous operations in a program. This is crucial in environments like Node.js, where asynchronous code (callbacks, promises, async/await) is heavily used. Without proper context tracking, it becomes difficult to debug, log, or maintain state across different parts of an application that rely on asynchronous execution.

In Node.js, **asynchronous context tracking** can be achieved through various mechanisms, most notably with **Async Hooks** (provided by the async\_hooks module). This module enables developers to track the lifecycle of asynchronous resources like timers, I/O operations, and promises, providing detailed control over what happens in each asynchronous operation.

**Key Concepts of Asynchronous Context Tracking**

1. **Execution Context**: This refers to the state or environment (such as variables, request data, etc.) that a particular piece of code runs in. In synchronous code, this is straightforward, but in asynchronous code, the execution context might be lost as the code yields to the event loop and resumes later.
2. **Context Propagation**: In asynchronous code, keeping track of the context across callbacks or promises can be tricky. Context propagation is the process of carrying execution context across different asynchronous operations.

**Tools and Techniques for Asynchronous Context Tracking**

**1. Async Hooks (Node.js)**

The async\_hooks module in Node.js allows you to hook into the lifecycle of asynchronous operations. It provides events that allow you to track the initialization, execution, and destruction of asynchronous resources.

The lifecycle events are:

* **init**: Triggered when a new async resource is initialized.
* **before**: Triggered just before the execution of an async callback.
* **after**: Triggered immediately after the async callback completes.
* **destroy**: Triggered when the async resource is destroyed.
* **promiseResolve**: Triggered when a promise is resolved.

**Example of Async Hooks for Context Tracking:**

js

Copy code

const async\_hooks = require('async\_hooks');

const fs = require('fs');

// A simple map to store context

const contexts = new Map();

// Function to track async context

const asyncHook = async\_hooks.createHook({

init(asyncId, type, triggerAsyncId) {

const parentContext = contexts.get(triggerAsyncId);

if (parentContext) {

// Propagate the parent context to the new async resource

contexts.set(asyncId, parentContext);

}

},

destroy(asyncId) {

// Remove the context when the async operation is destroyed

contexts.delete(asyncId);

}

});

// Enable the hook

asyncHook.enable();

// Example function to set and propagate context

function runWithContext(context, callback) {

const asyncId = async\_hooks.executionAsyncId();

contexts.set(asyncId, context);

callback();

}

// Using the function to track context

runWithContext({ user: 'Alice' }, () => {

setTimeout(() => {

const asyncId = async\_hooks.executionAsyncId();

console.log('Context:', contexts.get(asyncId)); // Outputs: { user: 'Alice' }

}, 100);

});

In this example:

* We create a hook using async\_hooks.createHook() that tracks the initialization and destruction of asynchronous resources.
* The context (in this case, user data) is propagated across the asynchronous boundaries so that it is available within the async callback.

**2. cls-hooked (Continuation Local Storage)**

Before the introduction of async\_hooks, many developers used libraries like **cls-hooked** to track asynchronous context. It provides a higher-level API for context tracking.

cls-hooked allows you to create "namespaces" that can store data and automatically propagate it through asynchronous boundaries.

Example with cls-hooked:

js

Copy code

const cls = require('cls-hooked');

const session = cls.createNamespace('my-session');

// Function to track context using cls-hooked

function runWithContext(context, callback) {

session.run(() => {

session.set('user', context.user);

callback();

});

}

// Example usage

runWithContext({ user: 'Bob' }, () => {

setTimeout(() => {

const user = session.get('user');

console.log('Current User:', user); // Outputs: 'Bob'

}, 100);

});

Here:

* cls-hooked automatically propagates the context (user in this case) through asynchronous boundaries (like setTimeout).

**3. AsyncLocalStorage (Built-in Replacement for cls-hooked)**

As of Node.js 12+, the AsyncLocalStorage API was introduced as a built-in replacement for cls-hooked. This API allows storing and propagating context through async operations.

Example with AsyncLocalStorage:

js

Copy code

const { AsyncLocalStorage } = require('async\_hooks');

const asyncLocalStorage = new AsyncLocalStorage();

// Function to run code with a specific context

function runWithContext(context, callback) {

asyncLocalStorage.run(context, callback);

}

// Example usage

runWithContext({ user: 'Charlie' }, () => {

setTimeout(() => {

const context = asyncLocalStorage.getStore();

console.log('Current Context:', context); // Outputs: { user: 'Charlie' }

}, 100);

});

In this example:

* AsyncLocalStorage.run() sets up a context that is propagated across asynchronous boundaries.
* Inside the setTimeout callback, the context can still be accessed.

**Use Cases for Asynchronous Context Tracking**

1. **Request/Response Context**: For web applications, you can track the context of each request throughout its lifecycle. This is useful for logging, auditing, or correlating errors with specific requests.
2. **Logging**: By attaching metadata (such as user ID or transaction ID) to the context, you can generate detailed logs that maintain this information across asynchronous calls.
3. **Error Handling**: Context tracking makes it easier to trace and debug asynchronous errors by associating error information with the relevant context.
4. **Performance Monitoring**: Tools like tracing and profiling benefit from context propagation, as it allows you to measure the time and performance of individual asynchronous operations while keeping track of the originating request or operation.

**Summary**

Asynchronous context tracking is critical for maintaining execution context across multiple asynchronous operations in Node.js. With tools like async\_hooks, AsyncLocalStorage, and cls-hooked, you can propagate and manage context efficiently, making it easier to debug, monitor, and enhance your Node.js applications.

# Buffers

In Node.js, the **Buffer** class is used to handle binary data directly, without needing to first convert it to a string. Buffers are useful for working with streams, binary files, and any raw data that needs to be handled efficiently, particularly in scenarios involving network communication or file I/O.

Buffers are part of the **buffer** module, which is included in Node.js's core, and they provide a way to work with raw binary data similar to arrays of integers but with a fixed size.

**Key Concepts of Buffers in Node.js**

1. **Fixed Size**: Buffers are allocated with a specific size in bytes, and this size cannot be changed after creation.
2. **Raw Data Storage**: Unlike strings, which are encoded in a character encoding like UTF-8, buffers store raw binary data. This allows efficient handling of I/O operations that involve binary files, network packets, and other low-level data streams.
3. **Encodings**: When converting buffers to strings or vice versa, you can specify an encoding. Common encodings include utf8, ascii, base64, hex, and latin1.

**Creating Buffers**

1. **From a String**: You can create a buffer from a string by specifying the string and encoding (default is UTF-8).

js

Copy code

const buf = Buffer.from('Hello, World!', 'utf8');

console.log(buf); // Prints: <Buffer 48 65 6c 6c 6f 2c 20 57 6f 72 6c 64 21>

1. **Allocating a Buffer**: You can create an empty buffer of a specified size (in bytes). This is particularly useful when you know the size of the data you're expecting.

js

Copy code

const buf = Buffer.alloc(10); // Creates a buffer of 10 bytes filled with zeros.

console.log(buf); // Prints: <Buffer 00 00 00 00 00 00 00 00 00 00>

1. **Buffer with Uninitialized Memory**: You can allocate a buffer with uninitialized memory for performance reasons, but this can lead to potential security risks as the buffer may contain old data.

js

Copy code

const buf = Buffer.allocUnsafe(10); // May contain random data

console.log(buf);

1. **From an Array of Bytes**: You can also create a buffer from an array of integers (representing bytes).

js

Copy code

const buf = Buffer.from([72, 101, 108, 108, 111]); // "Hello" in ASCII

console.log(buf.toString()); // Prints: Hello

**Common Buffer Operations**

1. **Writing to a Buffer**: You can write strings or binary data to a buffer.

js

Copy code

const buf = Buffer.alloc(20);

buf.write('Hello', 'utf8');

console.log(buf); // Prints: <Buffer 48 65 6c 6c 6f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00>

1. **Reading from a Buffer**: Convert the contents of a buffer back to a string.

js

Copy code

const buf = Buffer.from('Hello, World!', 'utf8');

console.log(buf.toString('utf8')); // Prints: Hello, World!

1. **Buffer Length**: The length of a buffer is its size in bytes.

js

Copy code

const buf = Buffer.from('Hello');

console.log(buf.length); // Prints: 5

1. **Concatenating Buffers**: Multiple buffers can be combined using Buffer.concat().

js

Copy code

const buf1 = Buffer.from('Hello, ');

const buf2 = Buffer.from('World!');

const buf3 = Buffer.concat([buf1, buf2]);

console.log(buf3.toString()); // Prints: Hello, World!

1. **Slicing a Buffer**: You can create a new buffer that references a portion of the original buffer. This operation is memory-efficient as the new buffer still references the same memory.

js

Copy code

const buf = Buffer.from('Hello, World!');

const slice = buf.slice(0, 5);

console.log(slice.toString()); // Prints: Hello

1. **Copying a Buffer**: You can copy the contents of one buffer into another buffer.

js

Copy code

const buf1 = Buffer.from('Hello');

const buf2 = Buffer.alloc(5);

buf1.copy(buf2);

console.log(buf2.toString()); // Prints: Hello

**Encoding and Decoding**

Buffers are often used for encoding or decoding data. For example, converting between text formats and binary formats such as Base64, hex, and binary is common.

* **Base64 Encoding**:

js

Copy code

const buf = Buffer.from('Hello, World!');

const base64 = buf.toString('base64');

console.log(base64); // Prints: SGVsbG8sIFdvcmxkIQ==

* **Hex Encoding**:

js

Copy code

const buf = Buffer.from('Hello, World!');

const hex = buf.toString('hex');

console.log(hex); // Prints: 48656c6c6f2c20576f726c6421

**Buffers in Streams**

Buffers are frequently used with Node.js streams (like file streams or network sockets). As streams emit chunks of data, they provide it as buffers. Here's a simple example of reading a file as a buffer:

js

Copy code

const fs = require('fs');

fs.readFile('example.txt', (err, data) => {

if (err) throw err;

console.log(data); // Prints the file contents as a buffer

});

**Buffer Pooling and Performance**

In high-performance applications, frequent buffer allocations can affect performance. To optimize buffer handling, Node.js employs a technique called **buffer pooling**, where small buffers (less than 8KB) are allocated from a shared memory pool. This reduces the overhead of allocating memory for each buffer.

For very large buffers, it's generally recommended to use Buffer.allocUnsafe() with caution, as it avoids initializing the buffer's memory, improving performance but potentially exposing old memory data.

**Security Considerations**

* **Uninitialized Buffers**: Buffers created with Buffer.allocUnsafe() are fast but can contain old data from the memory, which might lead to security vulnerabilities. Use Buffer.alloc() to create zero-filled buffers for safer memory allocation.
* **Data Validation**: Always validate inputs when working with buffers, especially when dealing with network or file I/O to avoid potential vulnerabilities like buffer overflows.

**Summary**

* **Buffers** in Node.js are essential for handling raw binary data.
* They are used heavily in network communication, file I/O, and streams.
* Buffers offer efficient data handling, especially when dealing with large volumes of data.
* Operations include reading, writing, slicing, concatenating, and encoding/decoding data.
* Buffer.alloc() and Buffer.from() are safer ways to create buffers, while Buffer.allocUnsafe() is faster but should be used with caution due to potential security risks.

# Streams

In Node.js, **streams** are objects that allow you to read or write data continuously, often in a more memory-efficient way, especially for large amounts of data. Instead of reading or writing everything at once, streams break data into smaller chunks, process it incrementally, and allow efficient management of resources.

Streams are a key part of Node.js because they allow for the handling of I/O-bound tasks like reading from files, network communication, or interacting with databases while keeping memory usage low.

**Types of Streams in Node.js**

There are four primary types of streams in Node.js:

1. **Readable Streams**: For reading data from a source (like a file or network socket).
2. **Writable Streams**: For writing data to a destination (like a file or HTTP response).
3. **Duplex Streams**: Streams that can both read and write (like TCP sockets).
4. **Transform Streams**: A special type of duplex stream that can modify or transform data as it is written and read (like gzip compression).

**Key Concepts**

* **Chunks**: Streams process data in chunks (usually Buffer or string objects).
* **Piping**: Data can be passed from one stream to another, creating a pipeline.
* **Backpressure**: This occurs when the destination stream cannot process the data as fast as the source provides it, and streams handle this situation efficiently.

**1. Readable Streams**

Readable streams are used to read data from a source. Some common examples of readable streams are file reading, HTTP request objects, or network socket reads.

You can read from a readable stream in two ways:

* **Event-driven**: Using events like data, end, and error.
* **Manual consumption**: Using the read() method to pull data from the stream.

**Example: Reading from a File Stream**

js

Copy code

const fs = require('fs');

const readableStream = fs.createReadStream('example.txt', { encoding: 'utf8' });

readableStream.on('data', (chunk) => {

console.log('Received chunk:', chunk);

});

readableStream.on('end', () => {

console.log('No more data.');

});

In this example:

* We are reading data from a file using the fs.createReadStream() method.
* The data event is triggered every time a chunk of data is available, and the end event signals the end of the stream.

**Example: Manual Consumption with read()**

js

Copy code

const fs = require('fs');

const readableStream = fs.createReadStream('example.txt');

readableStream.on('readable', () => {

let chunk;

while (null !== (chunk = readableStream.read())) {

console.log(`Received chunk: ${chunk}`);

}

});

readableStream.on('end', () => {

console.log('No more data.');

});

**2. Writable Streams**

Writable streams are used to write data to a destination. This can be a file, HTTP response, or any writable resource. Some common writable streams are fs.createWriteStream(), HTTP response objects, and network socket writes.

**Example: Writing to a File Stream**

js

Copy code

const fs = require('fs');

const writableStream = fs.createWriteStream('output.txt');

writableStream.write('Hello, World!\n');

writableStream.write('Writing more data...\n');

writableStream.end(); // Signals that we're done writing

writableStream.on('finish', () => {

console.log('Finished writing to the file.');

});

* The write() method writes chunks of data to the file.
* The end() method signals that no more data will be written to the stream, which also triggers the finish event.

**3. Duplex Streams**

Duplex streams are both readable and writable. A common example is a network socket, where you can send and receive data simultaneously.

**Example: Duplex Stream (TCP Socket)**

js

Copy code

const net = require('net');

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

socket.write('Hello from server!\n');

socket.on('data', (chunk) => {

console.log('Received data from client:', chunk.toString());

socket.write(`Echo: ${chunk}`);

});

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

console.log('Connection ended.');

});

});

server.listen(8080, () => {

console.log('Server is listening on port 8080');

});

In this example:

* The socket is a duplex stream.
* You can read data from the socket (via the data event) and also write data to it (via socket.write()).

**4. Transform Streams**

Transform streams are a type of duplex stream that can modify the data while reading and writing. A classic use case is compression (e.g., Gzip) or data encryption.

**Example: Transform Stream (Gzip Compression)**

js

Copy code

const fs = require('fs');

const zlib = require('zlib');

const readableStream = fs.createReadStream('example.txt');

const writableStream = fs.createWriteStream('example.txt.gz');

const gzip = zlib.createGzip();

readableStream.pipe(gzip).pipe(writableStream);

writableStream.on('finish', () => {

console.log('File successfully compressed.');

});

In this example:

* We read from a file, pipe the data through a gzip transform stream, and then write the compressed data to another file.

**Stream Methods and Events**

**Common Methods:**

* **stream.pipe(destination)**: Pipes data from the readable stream to the writable stream.

js

Copy code

readableStream.pipe(writableStream);

* **stream.unpipe(destination)**: Stops piping between streams.
* **stream.pause()**: Temporarily stops the flow of data.
* **stream.resume()**: Resumes the flow of data.
* **stream.write(chunk)**: Writes a chunk of data to a writable stream.
* **stream.end([chunk])**: Signals the end of writing to a writable stream.

**Common Events:**

* **data**: Fired when a chunk of data is available in a readable stream.
* **end**: Fired when no more data will be provided by the readable stream.
* **error**: Fired when there’s an error in the stream.
* **finish**: Fired when all data has been written to a writable stream.
* **readable**: Fired when the stream is ready to be read.

**Piping Streams**

Piping is one of the most powerful features of Node.js streams. It allows you to chain multiple streams together to process data. For example, you can read from a file, compress it, and then write the compressed data to a new file—all in one pipeline.

**Example: Piping Streams**

js

Copy code

const fs = require('fs');

const zlib = require('zlib');

const readableStream = fs.createReadStream('input.txt');

const writableStream = fs.createWriteStream('output.txt.gz');

const gzip = zlib.createGzip();

// Piping the readable stream into the gzip transform stream and then into the writable stream

readableStream.pipe(gzip).pipe(writableStream);

writableStream.on('finish', () => {

console.log('File successfully compressed.');

});

**Handling Backpressure**

When data is being produced faster than it can be consumed, backpressure occurs. Streams in Node.js handle backpressure automatically by pausing the source stream when the destination stream’s buffer is full, and resuming once there’s space available again.

For example:

* A writable stream's write() method returns false when its buffer is full.
* A readable stream will pause the flow of data until the writable stream signals it’s ready for more data.

**Summary**

* **Readable Streams**: For reading data in chunks from a source.
* **Writable Streams**: For writing data in chunks to a destination.
* **Duplex Streams**: For both reading and writing, such as TCP sockets.
* **Transform Streams**: Duplex streams that transform the data during reading and writing (e.g., compression).

# Cluster

In Node.js, **Cluster** is a module that allows you to scale an application by creating multiple instances (workers) of a Node.js process. This is particularly useful for taking advantage of multi-core systems, as a single instance of Node.js runs on a single thread by default.

By clustering, you can distribute incoming connections across multiple processes (workers), which can run on different CPU cores, improving the performance and resilience of your application.

**Why Use Clustering?**

1. **Single-Threaded Nature of Node.js**: By default, Node.js runs JavaScript in a single thread. While this is efficient for I/O operations (thanks to the event loop), it can become a bottleneck for CPU-bound tasks. Clustering helps in scaling an application by utilizing multiple CPU cores.
2. **Improved Throughput**: Cluster allows an application to handle more requests by balancing the load across worker processes, which are copies of the original application.
3. **Fault Tolerance**: If one worker crashes, others remain unaffected, and the master process can spawn new workers to replace crashed ones.

**How Clustering Works**

1. **Master Process**: The master process is responsible for forking worker processes. It doesn't handle incoming requests directly; instead, it manages and distributes them among workers.
2. **Worker Processes**: These are the actual processes that handle requests. Each worker is essentially a copy of the Node.js application.

**Example of Using Clustering**

Here’s a basic example of using the cluster module in Node.js:

js

Copy code

const cluster = require('cluster');

const http = require('http');

const os = require('os');

// Get the number of available CPU cores

const numCPUs = os.cpus().length;

if (cluster.isMaster) {

console.log(`Master ${process.pid} is running`);

// Fork workers (one per CPU core)

for (let i = 0; i < numCPUs; i++) {

cluster.fork();

}

// If a worker dies, replace it with a new one

cluster.on('exit', (worker, code, signal) => {

console.log(`Worker ${worker.process.pid} died`);

console.log('Starting a new worker...');

cluster.fork();

});

} else {

// Workers share the TCP connection and can handle requests

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

res.writeHead(200);

res.end(`Hello from Worker ${process.pid}\n`);

}).listen(8000);

console.log(`Worker ${process.pid} started`);

}

**How the Code Works:**

1. **Master Process**:
   * The master process spawns a worker for each CPU core using cluster.fork().
   * It listens for the exit event to detect when a worker dies, and it respawns a new worker.
2. **Worker Processes**:
   * Each worker runs the same server code and listens on the same port (8000 in this case). Incoming requests are automatically balanced between the workers.
   * The workers share the same TCP connection.

**Key Cluster Methods and Properties**

* **cluster.isMaster**: A boolean that is true if the process is the master process.
* **cluster.isWorker**: A boolean that is true if the process is a worker.
* **cluster.fork()**: Forks a new worker process.
* **cluster.on('exit', callback)**: Listens for when a worker process dies and executes a callback function.
* **cluster.workers**: An object containing all the worker processes currently running.

**Load Balancing in Clusters**

Node.js uses a round-robin approach to load balancing by default. In this setup, the master process listens on a TCP/IP port, and it distributes the incoming connections among the workers. However, there are two different ways Node.js implements load balancing:

1. **Round-Robin (Default on UNIX)**: The master process listens on the port and distributes incoming requests to workers in a round-robin fashion.
2. **Reversed Load Balancing (Default on Windows)**: On Windows, each worker listens on the server port, and the operating system’s load balancer is used to distribute connections.

**Cluster Event Handling**

Cluster has several events that can help you manage worker processes effectively:

* **online**: Triggered when a worker is successfully forked and ready.

js

Copy code

cluster.on('online', (worker) => {

console.log(`Worker ${worker.process.pid} is online`);

});

* **exit**: Triggered when a worker dies. This can help implement auto-restarting of workers.

js

Copy code

cluster.on('exit', (worker, code, signal) => {

console.log(`Worker ${worker.process.pid} died`);

cluster.fork(); // Auto-restart the worker

});

**Graceful Shutdown with Clusters**

Handling shutdowns gracefully is important, especially in production, to ensure that workers finish processing ongoing requests before being terminated.

**Example of Graceful Shutdown**

js

Copy code

if (cluster.isMaster) {

for (let i = 0; i < numCPUs; i++) {

cluster.fork();

}

cluster.on('exit', (worker, code, signal) => {

console.log(`Worker ${worker.process.pid} died. Restarting...`);

cluster.fork();

});

process.on('SIGTERM', () => {

console.log('Master is shutting down...');

for (const id in cluster.workers) {

cluster.workers[id].kill('SIGTERM');

}

process.exit();

});

} else {

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

res.writeHead(200);

res.end(`Handled by worker ${process.pid}`);

});

server.listen(8000);

process.on('SIGTERM', () => {

console.log(`Worker ${process.pid} is shutting down...`);

server.close(() => {

process.exit();

});

});

}

In this example:

* Workers are terminated gracefully when the master process receives a SIGTERM signal.
* Each worker finishes processing its ongoing requests before exiting.

**Benefits of Using Clustering**

1. **Utilization of Multiple CPU Cores**: Node.js is single-threaded by nature, but clustering allows the usage of multiple CPU cores by creating worker processes.
2. **Increased Throughput**: Multiple workers can handle more requests simultaneously, increasing the throughput of the application.
3. **Resilience**: If one worker crashes, others continue to run, and new workers can be spawned to replace the crashed ones, improving the overall availability of the application.

**Drawbacks**

1. **Shared Memory**: Workers don’t share memory. If you want to share data between workers, you need to use inter-process communication (IPC), a database, or an external store like Redis.
2. **Node.js Load Balancing**: The built-in load balancing is basic, and for more complex scenarios, a dedicated load balancer like Nginx or HAProxy may be more appropriate.

**Cluster vs. Child Processes**

While **cluster** allows you to create multiple worker processes to handle HTTP requests, you can also use the **child\_process** module to spawn separate processes for CPU-bound tasks. The primary difference is:

* **Cluster**: Used for scaling HTTP servers by creating workers that share the same port.
* **Child Processes**: Used for spawning additional processes to perform heavy computations or CPU-bound tasks in parallel.

# domain

The **domain** module in Node.js is a mechanism used to handle multiple asynchronous operations as a single group, particularly to manage error handling for asynchronous code. It helps in catching and managing errors across different asynchronous callbacks and promises, without having to explicitly pass error handlers in every function or using try-catch blocks for every async operation.

However, it’s worth noting that the domain module is deprecated in newer versions of Node.js and is discouraged for use in production. Node.js core team advises using more modern patterns, such as **Promises** with **async/await** or other error-handling mechanisms like **process.on('uncaughtException')** for uncaught errors, and **try-catch** with **synchronous code**.

**Why domain Was Introduced**

In earlier versions of Node.js, handling errors in asynchronous code was challenging. You had to manage error callbacks for each asynchronous function individually. The domain module was introduced to group related asynchronous I/O operations and handle errors in bulk. However, it had several limitations and inconsistencies in its behavior, leading to its deprecation.

**How the Domain Module Works**

A **domain** can be thought of as an execution context for handling multiple asynchronous operations and catching errors that occur within that context.

There are two ways to work with domains:

1. **Implicit Binding**: You bind a domain to all callbacks and I/O operations within the domain.
2. **Explicit Binding**: You explicitly bind certain callbacks or event emitters to the domain.

**Example of Using Domains**

Here’s an example of how the domain module works:

js

Copy code

const domain = require('domain');

const http = require('http');

// Create a domain

const d = domain.create();

d.on('error', (err) => {

console.error('Domain caught an error:', err);

});

// Run the server inside the domain

d.run(() => {

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

// Simulate an asynchronous error

setTimeout(() => {

throw new Error('Async error in server');

}, 1000);

}).listen(3000);

console.log('Server is listening on port 3000');

});

In this example:

* A domain is created with domain.create().
* The run() method executes code inside the domain.
* Any errors that occur inside asynchronous callbacks (like the setTimeout() in this example) are caught by the domain’s error event handler.

Without the domain module, handling such asynchronous errors across callbacks would require manually passing error handlers.

**Explicit Binding Example**

You can also explicitly bind event emitters or callbacks to a domain:

js

Copy code

const domain = require('domain');

const EventEmitter = require('events');

const d = domain.create();

const emitter = new EventEmitter();

d.on('error', (err) => {

console.error('Domain caught an error:', err);

});

// Explicitly bind the event emitter to the domain

d.add(emitter);

emitter.on('data', (data) => {

throw new Error('Error in event emitter');

});

emitter.emit('data', 'some data');

Here:

* We explicitly bind the EventEmitter instance to the domain using d.add(emitter).
* Any errors that occur in the event emitter’s listeners will be caught by the domain’s error handler.

**Limitations of Domains**

The domain module has several drawbacks and edge cases that led to its deprecation:

1. **Unpredictability**: Domains sometimes don’t catch all errors, and their behavior can vary depending on how asynchronous code is written.
2. **Performance**: Domains add an overhead in managing the error-catching mechanisms, which can impact the performance of your Node.js application.
3. **Deprecated**: As of Node.js v4.0.0, the domain module is deprecated, meaning it’s still available but should not be used in future projects.

**Modern Error Handling Alternatives**

Due to the limitations of the domain module, modern Node.js applications should rely on alternative approaches for error handling:

1. **try-catch with Async/Await**: This is now the preferred way to handle errors in asynchronous code using **Promises** and **async/await**. It allows for better readability and more predictable error handling.

js

Copy code

async function handleRequest(req, res) {

try {

const data = await asyncFunction();

res.end(data);

} catch (err) {

console.error('Caught an error:', err);

res.statusCode = 500;

res.end('Internal Server Error');

}

}

1. **Event Listeners on process**: For uncaught exceptions, you can use process.on('uncaughtException') or process.on('unhandledRejection') to catch and handle global errors.

js

Copy code

process.on('uncaughtException', (err) => {

console.error('Uncaught exception:', err);

});

process.on('unhandledRejection', (reason, promise) => {

console.error('Unhandled promise rejection:', reason);

});

1. **Error Handling in Express**: If you’re using Express.js or a similar web framework, you can use **middleware** to handle errors globally.

js

Copy code

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

console.error('Error:', err);

res.status(500).send('Internal Server Error');

});

# Zone

In Node.js, the concept of **zones** is not natively supported in the core modules, but it refers to a feature found in some asynchronous programming libraries and frameworks. Zones provide a way to maintain and track the execution context across asynchronous operations, which can be particularly useful for tasks like managing request-specific data or implementing error handling in complex asynchronous flows.

**What are Zones?**

**Zones** are a mechanism for maintaining context across asynchronous operations. They are often used to:

* **Track Execution Context**: Keep track of context-specific data (like user sessions or request-specific data) throughout asynchronous calls.
* **Error Handling**: Capture and manage errors within a specific context or execution flow.
* **Instrumentation**: Collect and monitor metrics or logs within a specific zone.

**Libraries Providing Zones**

Several libraries and frameworks offer zone-like functionality for managing execution context in JavaScript and Node.js:

1. **Zone.js**:
   * Zone.js is a library originally designed for Angular to provide context-aware execution across asynchronous operations. It allows for tracking and managing context across tasks, promises, and event handlers.
   * It is not natively included in Node.js but can be used in Node.js environments for context management.

To use Zone.js in a Node.js application:

bash

Copy code

npm install zone.js

Example usage in Node.js:

js

Copy code

const zone = require('zone.js');

// Create a new zone

const myZone = zone.fork({

name: 'myZone',

properties: {

requestId: '12345'

}

});

// Run code within the zone

myZone.run(() => {

console.log(zone.get('requestId')); // '12345'

setTimeout(() => {

console.log(zone.get('requestId')); // '12345'

}, 1000);

});

1. **Async Hooks**:
   * Node.js’s built-in async\_hooks module provides low-level hooks into the lifecycle of asynchronous operations. Although not exactly zones, it allows tracking asynchronous resources and maintaining context.
   * For context management similar to zones, you can use async\_hooks to implement custom solutions.

Example using async\_hooks:

js

Copy code

const async\_hooks = require('async\_hooks');

const fs = require('fs');

const asyncLocalStorage = new Map();

const hook = async\_hooks.createHook({

init(asyncId, type, triggerAsyncId, resource) {

const context = asyncLocalStorage.get(triggerAsyncId);

if (context) {

asyncLocalStorage.set(asyncId, context);

}

},

destroy(asyncId) {

asyncLocalStorage.delete(asyncId);

},

});

hook.enable();

async function example() {

asyncLocalStorage.set(async\_hooks.executionAsyncId(), { requestId: '12345' });

setTimeout(() => {

console.log(asyncLocalStorage.get(async\_hooks.executionAsyncId())); // { requestId: '12345' }

}, 1000);

}

example();

**Use Cases for Zones**

1. **Request Context Management**:
   * In web applications, you might want to maintain request-specific data (like user authentication details) throughout the entire lifecycle of a request, including across asynchronous operations.
2. **Error Tracking and Reporting**:
   * Zones can help in capturing errors that occur in asynchronous operations and reporting them with context-specific information.
3. **Instrumentation and Monitoring**:
   * Tracking execution context can be useful for performance monitoring, logging, and debugging purposes.