**Worker thread:**

**worker threads** are used for multithreading in Node.js. This avoids blocking the main event loop, allowing your application to remain responsive.

**Worker Threads Module**: The worker\_threads module in Node.js provides a way to create and manage worker threads.

**Main Thread vs Worker Thread**:

* The **main thread** is the primary execution thread of a Node.js application.
* A **worker thread** runs in isolation, has its own event loop, and cannot access variables from the main thread unless explicitly shared.

**Data Sharing**:

* Data can be passed between threads using **messages** (via postMessage and on('message')).
* For shared memory, you can use **SharedArrayBuffer** or Node.js **MessageChannel**.

worker.**postMessage**('Hello, Worker!');

Here **postmessage** is used to send a message to worker

**USES:**

 Tasks like image processing, encryption, compression, or mathematical calculations (e.g., matrix operations or factorials).

 Parsing or transforming large datasets (e.g., CSV or JSON files).

**workerpool in JavaScript**

workerpool is a third-party library for Node.js that simplifies the process of managing worker threads or processes

The **event loop** is the core mechanism in Node.js that enables non-blocking, asynchronous operations. It allows Node.js to handle multiple operations (like I/O tasks, timers, and more) without blocking the main thread, making it highly efficient for I/O-bound tasks.

console.log('start');

setTimeout(() => {

console.log('setTimeout');

}, 0);

setImmediate(() => {

console.log('setImmediate');

});

Promise.resolve().then(() => {

console.log('Promise');

});

process.nextTick(() => {

console.log('nextTick');

});

console.log('end');

**op:**

start

end

nextTick

Promise

setTimeout

setImmediate

**Event vs Eventemitter**

The **EventEmitter** class in Node.js provides a mechanism to create and manage custom events

An **event** is a signal that something has occurred.

const EventEmitter = require('events');

// Create an EventEmitter instance

const myEmitter = new EventEmitter();

// Define an event listener

myEmitter.on('greet', (name) => {

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

});

// Trigger the 'greet' event

myEmitter.emit('greet', 'Alice');

 .on(event, listener):

* Registers a listener for a specific event.

 .emit(event, [...args]):

* Triggers the event and invokes the associated listeners.

 .once(event, listener):

* Adds a listener that will be executed only once.

 .removeListener(event, listener):

* Removes a specific listener.

 .removeAllListeners([event]):

* Removes all listeners for a specific event or all events.

**Loadbalancing**

Distributes workload evenly across multiple servers.

**Streams**

Streams are a powerful feature that allows handling data in chunks rather than loading it all at once, which is efficient for both memory usage and performance.

1. **Writable:** We can write data to these streams. **e.g.,**fs.createWriteStream().
2. **Readable:** We can read data from these streams. **e.g.,** fs.createReadStream().
3. **Duplex:** Streams that are both, Writable as well as Readable. **e.g.,**net.socket.
4. **Transform:** Streams that can modify or transform the data as it is written and read. **e.g.,**zlib.createDeflate.

**Body parser:**

body-parser is essential for handling incoming data in a variety of formats, such as JSON, URL-encoded form data, and raw or text data. It transforms this data into a readable format under req.body for easier processing in your application.

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

// Parse application/json

app.use(bodyParser.json());

// Parse application/x-www-form-urlencoded

app.use(bodyParser.urlencoded({ extended: true }));

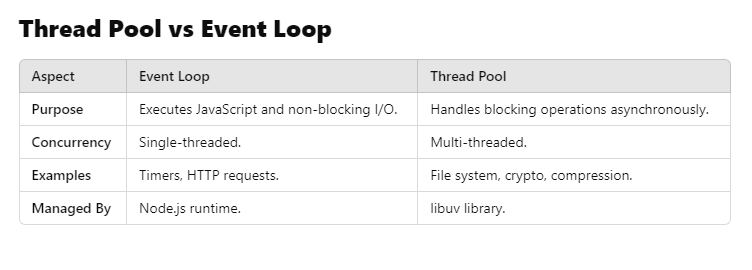
app.post('/submit', (req, res) => {

console.log(req.body); // Access parsed request body

res.send('Data received');

});

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



Approaches to Validation in Node.js:

* 1. Manual Validation
  2. Using Validation Libraries

 **Joi** (Highly flexible and feature-rich)

 **Yup** (Schema-based and lightweight)

 **Validator.js** (String validation utilities)

 **Express-Validator** (Middleware for Express.js)

* 1. Custom Middleware for Validation

**jwt has 3 parameters what are those**

<Header>.<Payload>.<Signature>

**how can we protect api's in nodejs**

**Authentication and Authorization**

* 1. Use JSON Web Tokens (JWT)
  2. OAuth2
  3. API Keys

2. Input Validation and Sanitization

3. Enable HTTPS

4. Use CORS for Cross-Origin Resource Sharing

5. Secure HTTP Headers

**Design patterns in nodejs** They help organize code, improve maintainability, and enhance scalability.

1. Creational Patterns
   1. Singleton
   2. Factory
2. Structural Patterns
   1. Module
   2. Proxy
3. Behavioral Patterns
   1. Observer
   2. Middleware
   3. Command
4. Asynchronous Patterns
   1. Callback
   2. Promise
   3. **Async/Await**
5. Concurrency Patterns
   1. Worker Threads
   2. Cluster

**Cluster module**

By default, a Node.js application runs on a single thread, meaning it can only use one CPU core at a time. The Cluster module spawns multiple processes, enabling the application to utilize multiple cores.

**Event-Driven Architecture**

Node.js follows an **event-driven programming model**, which allows the application to handle multiple operations asynchronously and efficiently. This model is integral to how Node.js handles I/O operations, making it lightweight and non-blocking.

**Event Loop**:

**Event Emitter**:

**Non-Blocking I/O**: