## Getting Started with [Node.js](http://node.js)

### **What is Node.js?**

Node.js is a runtime environment that allows you to run JavaScript on the server-side. Built on Chrome’s V8 JavaScript engine, Node.js is designed for building scalable network applications.

* Released: 2009 by Ryan Dahl
* Language Used: JavaScript
* Platform: Cross-platform (Windows, Linux, macOS)

### **Advantages of** [**Node.js**](http://node.js)

| **Feature** | **Benefit** |
| --- | --- |
| Non-blocking I/O | Handles many connections simultaneously, suitable for real-time systems |
| JavaScript on Server | Uses the same language for both frontend and backend |
| Single-threaded Model | Lightweight and efficient for I/O-heavy operations |
| Fast Execution | Uses Google V8 engine for high-performance JavaScript execution |
| Large Ecosystem | Has a huge number of packages (over 1.5M in npm registry) |
| Active Community | Constant improvements and support |
| Microservice Friendly | Suitable for building lightweight services |

### **Installation and Setup**

**Install** [**Node.js**](http://node.js)

Download from<https://nodejs.org>

Verify installation:

node -v

npm -v

### **Real time use-cases**

| **Industry/Domain** | **Use Case Example** |
| --- | --- |
| Chat Applications | Slack clone, WhatsApp web backend using WebSockets |
| Live Collaboration | Google Docs-like real-time document editing |
| Gaming | Multiplayer games with real-time state sync |
| Streaming Services | Video/audio streaming (Netflix-like) |
| IoT Systems | Devices sending real-time data to servers |
| FinTech | Real-time stock updates, trading dashboards |
| E-commerce | Inventory updates, live order status, real-time notifications |
| HealthTech | Monitoring patient vitals in real time |

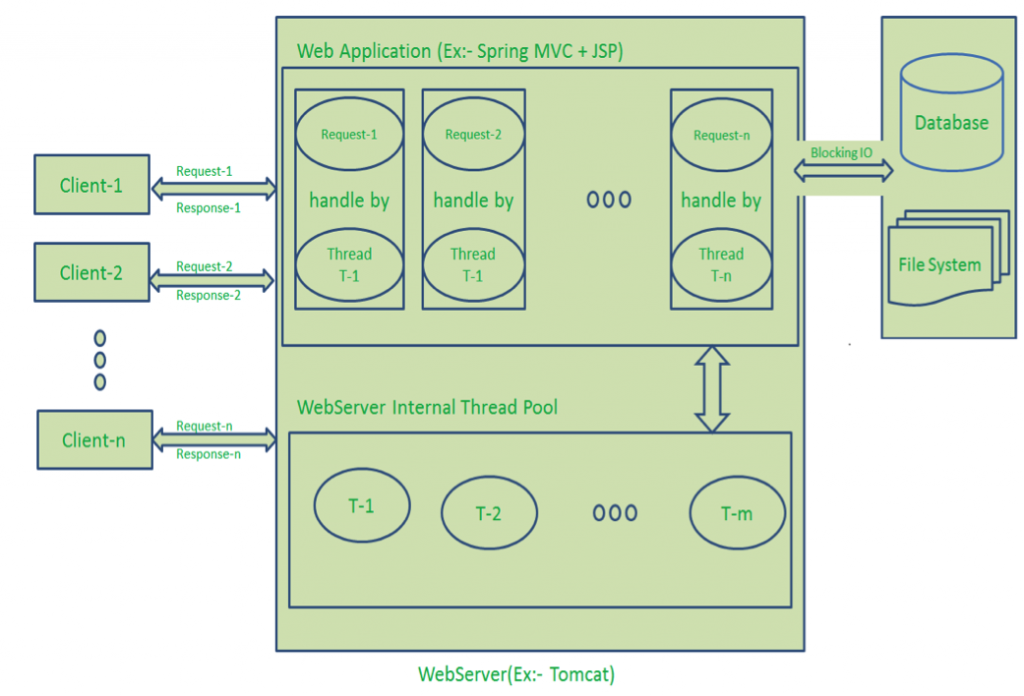
### Popular [Node.js](http://node.js) Libraries

| **Package** | **Use Case** |
| --- | --- |
| express | Web server framework |
| mongoose | MongoDB ODM |
| socket.io | Real-time communication |
| dotenv | Manage environment variables |
| cors | Enable Cross-Origin Resource Sharing |
| jsonwebtoken | JWT Authentication |
| nodemailer | Send emails from Node.js apps |

### 

### [**Node.js**](http://node.js) **Architecture**

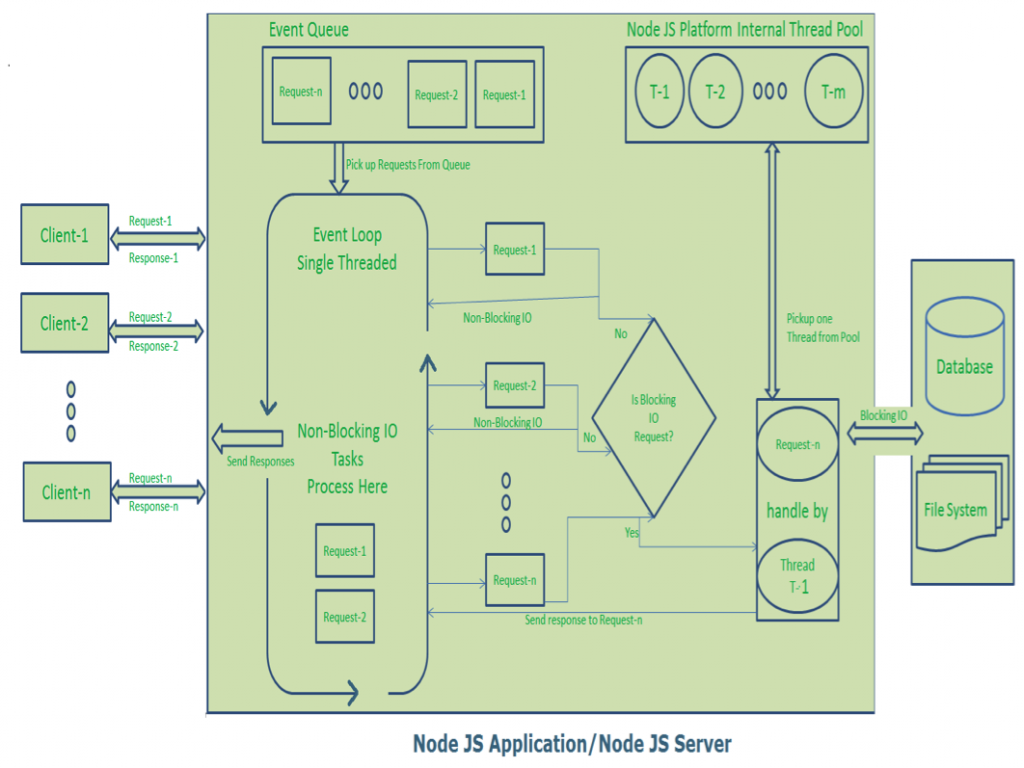
**Request/Response Model**



Drawbacks of Request/Response Stateless Model:

* Handling more and more concurrent client’s request is bit tough.
* When Concurrent client requests increases, then it should use more and more threads, finally they eat up more memory.
* Sometimes, Client’s Request should wait for available threads to process their requests.
* Wastes time in processing Blocking IO Tasks.

### **Single Threaded Event Loop**



**Advantages:**

1. Handling more and more concurrent client’s request is very easy.
2. Even though our Node JS Application receives more and more Concurrent client requests, there is no need of creating more and more threads, because of Event loop.
3. Node JS application uses less Threads so that it can utilize only less resources or memory

## Interactive node with REPL – **Read–Eval–Print Loop**

**REPL** stands for:

* **R**ead: Reads user input, parses it into JavaScript data structure.
* **E**val: Evaluates the data structure.
* **P**rint: Prints the result.
* **L**oop: Loops back to read again.

In simple terms, **REPL is a CLI (Command Line Interface)** tool that allows you to execute JavaScript code **interactively**, line by line, right in your terminal.

**command:** node and click enter

## Node.js Module System

In Node.js, **modules** are reusable pieces of code that encapsulate logic and can be **imported/exported** across files.

**Types of Modules**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Core Modules** | **Built into Node.js** | **fs, path, http** |
| **Local Modules** | **Custom modules created by developers** | **require('./utils.js')** |
| **Third-party Modules** | **Installed via npm** | **require('express')** |

#### **Core Modules:**

##### **fs (File System)**

Used to interact with the file system.

const fs = require('fs');

// Read a file

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

##### **Http**

Used to build HTTP servers and clients.

const http = require('http');

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

res.write('Hello World');

res.end();

}).listen(3000);

##### **Https**

Used for HTTPS (secure HTTP) servers.

const https = require('https');

https.get('https://jsonplaceholder.typicode.com/posts/1', res => {

res.on('data', d => process.stdout.write(d));

});

##### **Url**

URL resolution and parsing.

const url = require('url');

const parsed = url.parse('https://example.com/page?query=123');

console.log(parsed.hostname); // 'example.com'

console.log(parsed.query); // 'query=123'

##### **Querystring**

Parses and formats URL query strings.

const qs = require('querystring');

const obj = qs.parse('name=Sai&role=Instructror);

console.log(obj.name); // 'Sai'

**Assert**

Provides assertion testing (mostly for debugging/test purposes).

const assert = require('assert');

assert.strictEqual(2 + 2, 4); // Passes

## Creating First App in [Node.js](http://node.js)

Steps:

1. Create a new folder
2. Go to the folder
3. Npm init -y
4. Create [index.js](http://index.js)
5. Write code inside [index.js](http://index.js)
6. Run it using “node [index.js](http://index.js)”

### 

## File System in [Node.js](http://node.js)

Node.js provides a built-in module called **fs** for interacting with the file system.

Allows you to:

* Create, read, write, delete files and directories
* Work synchronously or asynchronously
* Handle file metadata

To use it:

const fs = require('fs');

#### **File Operations:**

#### 

##### **Reading Files:**

**Asynchronous (callback-based)**

const fs = require('fs');

fs.readFile("package.json",(err,data) =>{

if(err){

console.log("error reading file", err);

return;

}

if(data){

console.log(data.toString());

}

})

**Synchronous**

const fs = require('fs');

try{

const buffer = fs.readFileSync("package.json");

console.log(buffer.toString());

}

catch(error){

console.log("error", error.message);

}

**Reading an Image:**

const fs = require('fs');

try{

const buffer = fs.readFileSync("NodeJS.jpg");

console.log(buffer);

}

catch(error){

console.log("error", error.message);

}

##### **Writing Files:**

**Asynchronous**

const fs = require("fs");

fs.writeFile("user.txt", "Hello users", (err) =>{

if(err){

console.log(err);

}

else{

console.log("file written");

}

})

**Synchronous:**

const fs = require("fs");

fs.writeFileSync("user.txt", "Hello users");

**Append to a file:**

fs.appendFile('example.txt', '\nMore text...', (err) => {

if (err) throw err;

console.log('Content appended!');

});

##### **Read and Write Operation:**

const fs = require('fs');

try{

var data = fs.readFileSync("NodeJS.jpg");

console.log(data);

fs.writeFileSync("NewFile.jpg",data);

}

catch(error){

console.log(error);

}

##### **Deleting Files:**

**Asynchronous:**

const fs = require('fs');

fs.unlink("NodeJS.jpg", (err) =>{

if(err){

console.log(err);

}

else{

console.log("deleted");

}

})

**Synchronous**

fs.unlinkSync('NodeJS.jpg);

##### **Renaming Files:**

**Asynchronous**

const fs = require('fs');

fs.rename("NewFile.jpg", "Newest.jpg", (err) =>{

if(err){

console.log("error", err);

}

else{

console.log("renamed");

}

})

**Synchronous**

fs.renameSync('old.txt', 'new.txt');

##### **Directory Operations:**

**Create a Directory:**

**Asynchronous**

const fs = require('fs');

fs.mkdir("newfolder", (err) =>{

if(err)

{

console.log(err);

}

else{

console.log("created")

}

})

**Synchronous**

fs.mkdirSync('myFolder');

**Read Contents of a Directory**

const fs = require('fs');

fs.readdir(".", (err,files) =>{

if(err){

console.log(err);

}

else{

console.log(files);

}

})

**Delete a Directory**

**Asynchronous**

fs.rmdir('myFolder', (err) => {

if (err) throw err;

console.log('Directory deleted!');

});

## Streams

**What is a Stream?**

A Stream in Node.js is:

An abstract interface for working with streaming data.

Instead of reading or writing data all at once, streams **process data chunk by chunk**.

✅ Saves memory for large files  
 ✅ Enables faster processing  
 ✅ Great for reading/writing files, network calls, HTTP, etc.

Why Use Streams?

**Without streams:**

* You load the **entire data into memory**.
* Bad for large files or large network responses.

**With streams:**

* Handle data in small chunks.
* Start processing data before the entire resource is loaded.
* Less memory usage.

**Types of Streams**

Node.js provides four core stream types:

| **Type** | **Description** |
| --- | --- |
| **Readable** | You can read data from it. |
| **Writable** | You can write data into it. |
| **Duplex** | Both readable and writable |
| **Transform** | Duplex + transform data as it passes through. |

**Events in Streams**

Most streams are EventEmitters.

Key events:

**Readable Streams**

* data → When a chunk is available
* end → No more data
* error → Something went wrong
* close → Stream closed

**Writable Streams**

* drain → Ready for more data
* finish → All data written
* error → Error during write
* close → Stream closed

**Reading from a Readable Stream**

**Example: Read file using streams**

**createReadStream:**

const fs = require("fs");

const readStream =fs.createReadStream("gistfile1.txt", {

encoding: "utf8",

highWaterMark: 64 \* 1024 // 64kb chunks

});

readStream.on("data", (chunk) =>{

console.log("chunk:", chunk);

})

readStream.on("end", () =>{

console.log("streaming completed");

} )

readStream.on("error", (err) =>{

console.log(err);

})

**Writing to a Writable Stream**

const fs = require("fs");

const writeStream = fs.createWriteStream("newfile.txt");

writeStream.write('Hello, ');

writeStream.write('world!\n');

writeStream.end('Goodbye.\n');

writeStream.on("finish", () =>{

console.log("All writes are complete");

})

writeStream.on("error", (err) =>{

console.log("error occured",err);

})

**Duplex: Read & Write: Using .pipe()**

const fs = require("fs");

const readStream = fs.createReadStream("gistfile1.txt",

{

encoding: "utf8",

highWaterMark: 64 \* 1024 // 64kb chunks

}

);

const writeStream = fs.createWriteStream("file2.txt",

{

encoding: "utf8",

highWaterMark: 64 \* 1024 // 64kb chunks

}

);

readStream.pipe(writeStream);

readStream.on("end", () =>{

console.log("reading finished");

})

readStream.on("error", (err) =>{

console.log("error occured while reading", err.message);

})

writeStream.on("finish", () =>{

console.log("writing finished")

})

writeStream.on("error", (err) =>{

console.log("error occured while writing:",err);

})

## Events

#### **What are Events in** [**Node.js**](http://node.js)**?**

Node.js is built around asynchronous, non-blocking I/O. Instead of waiting for operations like file reads or network requests to finish, Node.js uses events to signal when things happen.

Think of events as signals or notifications:

* “Hey! The file is done loading.”
* “A new HTTP request arrived.”
* “A timer has expired.”

The **EventEmitter** class in Node.js (from the events module) makes it easy to **listen for** and **emit** custom events in your programs.

Node.js provides a built-in module called events:

const EventEmitter = require('events');

#### **Example**

const events = require("events");

const EventEmitter = new events.EventEmitter();

EventEmitter.on("connect", function(){

console.log("user connected");

})

EventEmitter.on("disconnect", function(){

console.log("user disconnected");

})

EventEmitter.emit("connect");

EventEmitter.emit("disconnect");

#### **Data Processing in events:**

const events = require("events");

const EventEmitter = new events.EventEmitter();

EventEmitter.on("connect", function(data){

console.log("The user '" +data.username+"' connected");

})

EventEmitter.on("disconnect", function(data){

console.log("The user '" +data.username+"' disconnected");

})

EventEmitter.emit("connect",{username: "kiran"});

EventEmitter.emit("disconnect",{username: "kiran"});

#### **The once Method**

const events = require("events");

const EventEmitter = new events.EventEmitter();

EventEmitter.once("init", function(){

console.log("server initiated");

})

EventEmitter.emit("init");

#### **Where are Events Used in** [**Node.js**](http://node.js)**?**

Events power many Node.js core features:

* **HTTP servers** → listen for 'request' events.
* **Streams** → listen for 'data', 'end', 'error'.
* **File system watchers** → listen for 'change'.
* **Child processes** → listen for 'exit'.

const http = require('http');

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

res.end('Hello World!');

});

server.on(**'request'**, () => {

console.log('New request received!');

});

server.listen(3000, () => {

console.log('Server running on port 3000');

});

#### **Benefits of Events in Node.js**

✅ Non-blocking:

* No waiting around. Your app can handle many things concurrently.

✅ Decoupling:

* Code is modular. Emitters and listeners don’t need to know about each other’s internals.

✅ Scalability:

* Handle many requests, data streams, or messages.

## 

## Web servers with http module

### **What is the http Module in** [**Node.js**](http://node.js)**?**

Node.js has a built-in module called http. It allows you to:

* **Create web servers** that listen for HTTP requests (GET, POST, etc.)
* Send responses back to clients (like browsers)
* Build REST APIs
* Serve HTML, JSON, or any other content over the web

Node.js’s http module is **low-level**, giving you complete control over request/response handling.

const http = require('http');

#### **Example 1 — A Simple HTTP Server**

const http = require('http');

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

res.write('Hello, World!');

res.end();

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000/');

});

✅ **How it works:**

* http.createServer() creates the server.
* The callback (req, res) is fired for every incoming request.
* res.write() sends data to the client.
* res.end() finishes the response.
* server.listen(3000) starts listening on port 3000.

#### **Example 2 — Sending HTML Response**

const http = require('http');

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

res.writeHead(200, { 'Content-Type': 'text/html' });

res.write('<h1>Welcome to My Website</h1>');

res.end();

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000/');

});

#### **Example 3 — Routing**

const http = require("http");

var server = http.createServer(function(request, response){

if(request.url == "/"){

response.write("<h1>Welcome to Node.JS</h1>");

response.end();

return;

}

if(request.url == "/about"){

response.write("<h1>Welcome to About </h1>");

response.end();

return;

}

if(request.url == "/careers"){

response.write("<h1>Welcome to Careers </h1>");

response.end();

return;

}

else{

response.write("<h1> This route is not available</h1>");

response.end();

return;

}

})

server.listen(9011, function() {

console.log("server started!!");

})

## Understanding clusters and working threads in nodejs

### The “Single-Threaded” Nature of Node.js

By default:

* Node.js runs JavaScript code in a single thread (the main thread).
* It handles many concurrent connections via non-blocking I/O.
* But… CPU-heavy tasks (e.g. calculations, JSON parsing) can block the event loop and slow everything down.

Hence:

* For multi-core CPU usage → use Clusters.
* For running CPU-heavy JS code off the main thread → use Worker Threads.

#### **Clusters in Node.js**

Node.js apps run in a single process. So no matter how many CPU cores your machine has, only **one core** is used.

✅ **Clusters** let you:

* Fork **multiple Node.js processes**.
* Each process runs its own instance of your server.
* They can share the same server port.
* This allows Node.js to fully utilize **multi-core CPUs**.

#### How Clustering Works

1. A **master process** launches **worker processes** (copies of your app).
2. Each worker:  
   * Handles client requests.
   * Runs independently of others.
3. The operating system load-balances requests among the workers.

#### Real-Life Cluster Example

#### 

Let’s say we want to:

* Use all CPU cores.
* Log which worker handled each request.

const cluster = require('cluster');

const http = require('http');

const os = require('os');

const numCPUs = os.cpus().length;

if (cluster.isMaster) {

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

// Fork workers.

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

cluster.fork();

}

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

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

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

cluster.fork();

});

} else {

// Workers run this HTTP server

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

res.writeHead(200);

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

}).listen(3000);

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

}

## [**Express.js**](http://express.js)

**Express.js** (often simply called **Express**) is a **web application framework for Node.js.**

Think of it as a set of tools and shortcuts that make building web servers and APIs in Node.js **much easier and faster.**

### Why Use Express.js?

✅ **Handles HTTP requests easily** (like GET, POST, PUT, DELETE).  
 ✅ Helps you build **REST APIs** or web apps quickly.  
 ✅ Lets you define routes for different URLs.  
 ✅ Makes it simple to work with data (like JSON).  
 ✅ Has lots of plugins (called “middleware”) to add extra features.

How to Install Express:

npm install express

#### **Simple Example**

const express = require('express');

const app = express();

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

res.send('Hello World!');

});

app.listen(3000, () => {

console.log('Server running on http://localhost:3000');

});

### What is Express Used For?

* Building **websites** (HTML pages)
* Creating **RESTful APIs**
* Serving static files (images, CSS, JS)
* Handling form data
* Adding security features
* Connecting to databases

CRUD App using [Express.js](http://express.js):

const express = require('express');

var app = express();

app.use(express.json());

var users= [];

app.listen(9011, () => console.log("server started!"));

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

res.send(users)

})

app.post("/user", (req,res) =>{

var username = req.body.username;

var address = req.body.address;

users.push({username,address});

res.send("user added");

})