Node.js is a JavaScript runtime that lets you run JavaScript outside the browser, powered by the V8 engine.

Chrome’s V8 engine is the high-performance JavaScript engine built by Google that powers Chrome and Node.js. It takes JavaScript code, compiles it directly into fast machine code using just-in-time (JIT) compilation. It also includes a highly tuned garbage collector that manages memory automatically.

non-blocking architecture: a runtime that handles many tasks without ever freezing itself in place.
When you ask it to perform something slow, it hands the job to the operating system and immediately returns control to your program.
The slow operation will report back later through a callback, a promise, or an async/await continuation.
Event-driven describes how Node knows when to resume your work. Its core is the event loop.

The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that a single JavaScript thread is used by default — by offloading operations to the system kernel whenever possible. Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.

Phases of event loop: Each phase has a FIFO queue of callbacks to execute. when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed.

When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

timers
pending callbacks
idle/prepare
poll <--- incoming: connection, data, etc.
check
close callbacks.

timers: this phase executes callbacks scheduled by setTimeout() and setInterval().
This is where callbacks from setTimeout and setInterval live. Node checks which timers are “ready”—meaning their delay has expired—and runs their callbacks. The delays aren’t perfectly precise; they’re minimum delays. If the loop is busy, your timer waits politely until this

phase comes around again.

pending callbacks: executes I/O callbacks deferred to the next loop iteration.
This is the “leftover paperwork” room. Some system-level callbacks—mostly from operations started in Node internals—land here when they couldn’t run earlier. These aren’t your usual fs or network callbacks. Think of it as Node’s own inbox for miscellaneous OS notifications.

idle, prepare: only used internally.
This one is mostly invisible to mortal developers. Node uses it internally to get ready for the next big phase (polling). You can imagine tiny librarians sorting future tasks and preloading metadata. No user-accessible callbacks run here.

poll: retrieve new I/O events; execute I/O related callbacks. This is the heart of the loop. Node checks for new I/O events: incoming TCP connections, arriving data packets, finished filesystem reads, DNS results, and so on.
If callbacks for these events are ready, they run here.
If nothing is ready, Node may actually wait (block) for new events to appear—unless timers or setImmediate callbacks are waiting in other phases. This part explains why Node excels at network servers: the poll phase continuously scoops up fresh I/O.

check: setImmediate() callbacks are invoked here.
This is the special nook reserved for setImmediate() callbacks. They always run here. The name sounds paradoxical—“immediate” but not “now”—yet the idea is simple: once the poll phase finishes its job, the check phase runs all queued immediate callbacks.

close callbacks: some close callbacks, e.g. socket.on('close', ...).
When something closes—like a TCP socket firing a 'close' event with a callback—that callback runs here. It’s the cleanup phase. Anything that has ended its lifecycle says farewell at this stage.

Node lives on the server side. It has direct access to the operating system: files, networking, processes, the filesystem, cryptography, threads (via worker threads), and sockets. You can read and write disk data, spawn child processes, manage servers—things a browser is never allowed to do, because browsers protect users from dangerous operations.

Node uses CommonJS modules (require), though modern builds support ES modules.
Browsers naturally use ES modules (import/export).
Node’s global object is global, while the browser’s global object is window.

npm is the package manager for Node.js. It’s a registry plus a tool. The registry stores millions of reusable packages;

CommonJS (CJS) is Node's older module system. It loads modules at runtime using require(). The export object is mutable and behaves like handing someone a toolbox that can be modified before use. This pattern is synchronous: when the VM hits a require, the module file must load immediately. Browsers couldn't support this originally, which is why bundlers were often needed.

One of the key features of the CommonJS module system is that it is synchronous. When you require a module, Node.js loads and executes the module immediately, blocking further execution until it completes.

```
// math.js
module.exports.add = (a, b) => a + b;

// index.js
const math = require('./math');
console.log(math.add(2, 3));

module.exports = {
greet,
multiply,
};
```

ES Modules (ESM) use import and export syntax. The structure is static—imports are known before code runs, which lets engines optimize and tree-shake (remove unused code). Browsers support ESM natively, so the same files can run in Node and the browser if written carefully. The loading is asynchronous and "hoisted," meaning imports behave like declarations at the top.

```
export function add(a, b) {
return a + b;
}

// index.js
import { add } from './math.js';
console.log(add(2, 3));
```

To accept input from the command line inside a Node program, the process object gives you process.argv, an array of arguments. The first entry is the Node binary path, the second is the script path, and everything after that is whatever the user typed.

process.argv is an array that contains the command-line arguments passed when you run a Node program.

process.argv stands for process arguments vector.

process.argv[0]: Path to the Node executable (e.g., /usr/local/bin/node)

process.argv[1]: Path to your script file (e.g., /home/me/app.js)

process.argv[2]: and onward Actual arguments you pass

```
const args = process.argv.slice(2);
node test.js apple banana
```

To read environment variables, Node exposes process.env, which is basically a dictionary of key–value pairs supplied by the system. These variables often hold secrets, config flags, modes, or API keys you don't want in your source code.

```
const port = process.env.PORT;
```

HTTP is the protocol the web uses for talking—almost like a set of polite rules for how browsers and servers exchange requests and responses.
A browser sends a request (like "give me /home.html"), and the server replies with content plus metadata (headers, status codes, cookies, etc.). Every click, every form submit, every image fetch is just more HTTP.
A Node.js server is simply a program that listens for these HTTP requests and responds to them. Node's http module gives you low-level control: you decide what to do when someone knocks on your virtual door.

data (when some data chunk arrives)
end (when all data finished loading)
error (if something goes wrong)
close (connection closed)

EventEmitter is a core Node.js class that allows objects to:
emit events
listen to events

```
const EventEmitter = require("events");
const emitter = new EventEmitter();
emitter.on("hello", () => {
console.log("Hello event triggered!");
});
emitter.emit("hello");  // triggers the event
```

' on() registers a listener
' emit() triggers an event

In HTTP requests, data arrives in chunks, not all at once. So Node.js uses events to notify you:
When a chunk of body data arrives !’ data event
When the entire body has arrived !’ end event

req.on('data') means “when the request receives data, run this code”.
req.on('end') means “when all the data is finished, run this code”.

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

const server = http.createServer((req, res) => {
res.writeHead(200, { "Content-Type":"text/plain" });
res.end("Hello from Node server!");
});

server.listen(3000, () => {
console.log("Server running at http://localhost:3000");
});
```

When a request arrives, Node gives you two objects:

A request object typically refers to the object that contains all the information about an HTTP request made by a client (such as a browser or an API client) to a server

req (IncomingMessage): holds details about the request—URL, HTTP method, headers, any body data.

```
req.method // GET
```

req.url // The full URL (including any query parameters). If you want to break this down into the path and query parameters, you can use the url module from Node.js.

```
'/users?id=123&name=John'

const parsedUrl = url.parse(req.url, true);
console.log(parsedUrl.pathname);  // '/users'
console.log(parsedUrl.query); // { id: '123', name: 'John' }
```

req.headers is a plain object containing all the header keys and values. It’s Node’s way of

exposing HTTP metadata in an easily inspectable form:

```
console.log(req.headers["content-type"]);
console.log(req.headers["authorization"]);
```

Body: The body is typically sent as JSON, form data, or plain text

```
let body = '';

// Collect data chunks from the request
req.on('data', chunk => {
body += chunk;
});

// When the request is fully received, handle the body
req.on('end', () => {
console.log('Request Body:', body);

// If it's JSON, parse it
try {
const parsedBody = JSON.parse(body);
console.log('Parsed Body:', parsedBody);
} catch (error) {
console.log('Error parsing JSON:', error);
}

res.end('Request body processed!');
});
```

1. Query Parameters (req.query)

- Used in the URL
- Typically used in GET requests
- Visible in browser URL
- Small amount of data
- Not suitable for sensitive data (unless HTTPS)

res (ServerResponse): your handle to craft a response—status code, headers, and body.
Writable Stream -> This means the server can write data back to the client.

Using res, you control:

status code
response headers
response body
when the response ends

```
res.statusCode = 200;             // set status
res.setHeader("Content-Type","text/plain"); // set headers
res.write("Hello ");              // write partial response
res.end("world!");                // finish response
```

```
res.setHeader("Content-Type","text/plain");
res.end("Hello from Node.js!");
```

```
res.setHeader("Content-Type","text/html");
res.end("<h1>Welcome</h1><p>This is HTML content</p>");
```

```
res.setHeader("Content-Type","application/json");
const user = { name: "John", age: 30 };
res.end(JSON.stringify(user));
```

```
res.setHeader('Content-Type','text/plain');
res.statusCode = 200;
res.write(data)
res.writeHead(statusCode, headers)
res.end(data); finishes the stream (must be called)
```

Path Module

It allows you to manipulate and resolve file paths in a way that's independent of the operating system, ensuring your application works consistently across different platforms

path.join([path1], [path2], [...paths])
Joins multiple path segments together into a single path, normalizing it to handle slashes appropriately.

```
const filePath = path.join('folder', 'subfolder', 'file.txt'); // 'folder/subfolder/file.txt'
```

path.resolve([from], [...to]) -> absolute path
Resolves a sequence of paths into an absolute path.

```
path.resolve('folder', 'subfolder', 'file.txt') // '/home/user/folder/subfolder/file.txt'
```

path.normalize(path)
Normalizes the given path by removing redundant . and .. segments and fixing any incorrect slashes.

```
path.normalize('/foo/bar//baz/asdf/quux/..'); // '/foo/bar/baz/asdf'
```

path.extname(path)
Returns the file extension of a given path.
This will return the extension part of the path, including the leading dot (.), or an empty string if there is no extension.

```
path.extname('index.html'); // Outputs: '.html'
```

path.basename(path, [ext]) -> gives file with extension, [ext] will remove the extension
Returns the last portion of a path (the file name). Optionally, you can provide an extension (ext) to remove it from the result.

```
path.basename('/foo/bar/baz/asdf/quux.html')); // Outputs: 'quux.html'
path.basename('/foo/bar/baz/asdf/quux.html', '.html')); // Outputs: 'quux'
```

path.dirname(path)
Returns the directory name of a path, removing the last portion (file name).

```
path.dirname('/foo/bar/baz/asdf/quux.html')); // Outputs: '/foo/bar/baz/asdf'
```

path.parse(path)
Parses a path into an object with properties: root, dir, base, ext, and name.

```
const parsedPath = path.parse('/home/user/dir/file.txt');
console.log(parsedPath);
// Outputs:
// {
//   root: '/',
//   dir: '/home/user/dir',
//   base: 'file.txt',
//   ext: '.txt',
//   name: 'file'
// }
```

path.format(pathObject)
Takes an object of path components and formats it into a path string.

```
const pathObject = {
root: '/',
dir: '/home/user/dir',
base: 'file.txt',
ext: '.txt',
name: 'file'
};
console.log(path.format(pathObject)); // Outputs: '/home/user/dir/file.txt'
```

path.join(dirname, "public", "index.html"):
dirname is a global variable that gives the directory name of the current module.
This line combines dirname, "public", and "index.html" into a single path string. This ensures that the file path works correctly across platforms.

Content-type and CORS sit right at the doorway where servers greet browsers. They decide what’s being sent and who’s allowed to receive it.

Content-Type is an HTTP header that tells the browser what kind of data the server is sending. Without it, the browser is like, “Is this text? JSON? A JPEG wearing a disguise?”

CORS—Cross-Origin Resource Sharing—is the browser’s bouncer. Browsers block scripts from making requests to different origins by default. An origin is a combination of protocol, domain, and port.
If your frontend at http://localhost:3000 tries to call a backend at http://localhost:5000, the browser stops it unless the server explicitly says, “Yes, that origin is welcome.”
CORS isn’t for security between servers; it’s strictly a browser rule meant to protect users.

The server signals this using CORS headers.

```
res.writeHead(200, {
"Access-Control-Allow-Origin": "*"
});

res.writeHead(200, {
"Access-Control-Allow-Origin": "http://localhost:3000",
"Access-Control-Allow-Methods": "GET, POST",
"Access-Control-Allow-Headers": "Content-Type"
});
```

HTTP Methods:

GET — fetch something.

POST — send data to the server.
PUT — replace something.
PATCH— update something partially.
DELETE — remove something.

Proxy
A proxy server acts as an intermediary between a client and another server. It can serve various purposes such as privacy, caching, security, or filtering.

The client sends a request to the proxy server.
The proxy server forwards the request to the target server.
The target server responds to the proxy, which then sends the response back to the client.

Types of Proxies:

1. Forward Proxy
Sits in front of clients.
Used to control access, hide client IP, or cache responses.

2. Transparent Proxy
Does not modify requests/responses.
Usually used for caching or monitoring.

3. Anonymous Proxy
Hides the client IP from the server.

Client knows about the proxy.
Server usually doesn't know the proxy is in between.

Reverse Proxy
A reverse proxy is a server that sits in front of one or more backend servers and forwards client requests to them. Essentially, it's the opposite of a forward proxy.

Client sends a request to the reverse proxy.
Reverse proxy decides which backend server should handle the request.
Response comes back via the reverse proxy to the client.

Load balancing: Distribute requests among multiple backend servers.
SSL termination: Handle HTTPS connections at the proxy instead of each backend server.
Caching: Reduce load on backend servers.
Security: Hide the internal server structure from external clients.

Forward proxy hides clients.
Reverse proxy hides servers.

SSL Termination
SSL termination (also called TLS termination) is the point where encrypted HTTPS traffic is decrypted back into plain HTTP.
This typically happens at a load balancer, reverse proxy, or edge device.

A user visits your site via HTTPS.
The load balancer or proxy decrypts the traffic — this is the “termination.”
The decrypted traffic is then forwarded to your backend servers (often via HTTP).
The response gets encrypted again before being sent back to the user.

An SSL certificate (more accurately a TLS certificate) is a digital file installed on a server that enables HTTPS. It proves the server’s identity and allows encrypted communication between a browser and a website.

Verify that the website really belongs to the domain it claims.
Encrypt traffic between the user and the server.

A CDN is a global network of distributed servers (“edge servers”) located close to users.
Its job: deliver content faster, reduce load on origin servers, and increase reliability.

CDN handles:

Caching static assets (HTML, CSS, JS, images, videos)
Security (DDoS protection, bot filtering, WAF)
TLS termination
Routing traffic efficiently
Reducing latency with closer servers
Sometimes dynamic content acceleration (Network optimization)

Your browser asks DNS:
"Where is www.example.com -> located?"

DNS returns an IP owned by the CDN
www.example.com --> 104.21.16.79 (Cloudflare Edge IP)
So the browser doesn’t even know the origin server IP.
It only sees the CDN edge IP.

Browser connects to the CDN edge server
Because DNS told it so.

It sends:
TCP/UDP handshake
TLS handshake
HTTP request
All to the edge server nearby, not the origin.

GeoDNS
Returns the IP of the location closest to the user.

CDNs hide the origin IP.

The origin usually:
Accepts traffic only from CDN IP addresses
Blocks all public traffic
Is unreachable directly from the internet

One IP can host many sites !’ server needs the domain (Host header / SNI)

A webhook is a simple, automatic way for one application to send real-time information to another application as soon as something happens. Instead of one system repeatedly checking (“polling”) for updates, the other system pushes data immediately when an event occurs.

A webhook is an HTTP request that one system sends to a URL you provide when a specific event occurs.
Event happens !’ Data is sent !’ Your system reacts automatically.

A webhook is:
A real-time event notification system
A simple HTTP POST request

Reverse proxy mainly focuses on network-level routing.
What it does:

Forwards HTTP requests to backend servers
Load balances (round-robin, least connections, etc.)
Caches static content
SSL termination (HTTPS !’ internal HTTP)
Rate limiting / basic security
Hides backend server IPs

An API Gateway does everything a reverse proxy does — BUT adds many API-specific features, especially for microservices. It controls, secures, and transforms API calls

What it does:

Authentication (JWT, OAuth2)
Authorization (Roles, permissions)
Rate limiting per user/client
Validates JWT token
Checks API key

API Gateway
An API Gateway is a specialized type of reverse proxy specifically for APIs. It provides a single entry point for clients to interact with multiple microservices.

Request Routing: Direct requests to correct microservices.
Authentication & Authorization: Validate API keys or tokens.
Rate Limiting & Throttling: Prevent abuse by limiting requests.
Caching: Reduce repeated requests to microservices.
Load Balancing: Similar to reverse proxy.

Reverse proxy mainly focuses on load balancing and security.
API Gateway focuses on microservice orchestration, routing, and API management.

Stream:
A stream is a way to handle data chunk-by-chunk instead of all at once.
Node.js streams provide an efficient way to handle data that isn’t available all at once—like files, network responses.
Instead of reading or writing data all at once, streams allow you to read or write data piece by piece, which is more memory-efficient and faster for large data.

Readable
Stream you can read data from
Reading files, HTTP requests

Writable
Stream you can write data to
Writing to files, HTTP responses

```
fs.createReadStream('in.txt')
.pipe(fs.createWriteStream('out.txt'));
```

Duplex
Stream that is both readable and writable
TCP sockets

Transform
Duplex stream that can modify data
Compressing or encrypting data

Memory efficiency: Don’t need to load the entire file in memory.
Speed: Start processing data as it comes.
Pipelining: You can chain streams together, e.g., read !’ transform !’ write.

A Readable Stream specifically represents a source of data that you read from.
fs.createReadStream()
http.IncomingMessage

1. fs.createReadStream() creates a Readable Stream that reads data from the filesystem.
const stream = fs.createReadStream(path, options);
const readableStream = fs.createReadStream('example.txt', { encoding: 'utf8' });
It streams file data in chunks rather than loading the entire file into memory.

```
const stream = fs.createReadStream('file.txt');
stream.on('data', chunk => {
console.log('chunk:', chunk.toString());
});
stream.on('end', () => console.log('done'));
```

2. http.IncomingMessage is what you receive in:
HTTP server: (req, res) !’ req is IncomingMessage

```
http.createServer((req, res) => {
req.on('data', chunk => console.log('body:', chunk.toString()));
req.on('end', () => res.end('ok'));
});
```

Backpressure = when the consumer is slower than the producer.
readable.pipe(writable);

pipe handles backpressure automatically

```
for await (const chunk of stream) {
console.log(chunk);
```

}
Clean and handles backpressure automatically.

createReadStream creates a readable stream for a file.
data event triggers when a chunk of data is available.
end event triggers when all data has been read.

A writable stream is a data-handling interface in Node.js that allows you to write data to a destination in small chunks, instead of sending it all at once. Handles backpressure

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

writableStream.write('Hello ');
writableStream.write('World!\n');
writableStream.end(); // Signals that writing is complete
writableStream.on('finish', () => {
console.log('Writing finished!');
});
```

You can process data as it arrives, Even if the data is small now, it might come from a slow or incremental source:
network request responses
incoming data from sensors
file uploads

Non-stream APIs typically buffer everything before writing. Streams write incrementally and prevent blocking the event loop.

The writable stream (res) API includes:

```
http.createServer((req, res) => {
res.write("Hello ");
res.write("world!");
res.end(); // finish will be emitted after this

res.on('finish', () => {
console.log("Response fully sent to client!");
});
}).listen(3000);
```

write()

end()
on('finish', ...)
backpressure handling
internal buffering

finish = Node has flushed all data to the underlying socket and the response is done.
& þNote:
finish means the server finished sending, not that the client received it.

Piping Streams
The most powerful feature of Node.js streams is piping. You can connect a readable stream to a writable stream easily.

readable.pipe(writable); Internally, Node.js handles buffering and memory efficiently.

A duplex stream is a stream that is both readable and writable at the same time, using separate read and write channels.
A duplex stream is a stream that you can write to and read from, and reading does not depend on writing (they act independently).

Transform Streams
Transform streams read, modify, and write data.

data Emitted when a chunk of data is available
end Emitted when no more data is available
error Emitted on stream error
finish Emitted when all data is flushed (writable)
close Emitted when stream closes

In Node.js, almost everything is event-driven. The EventEmitter class is at the core of this system. It allows an object to emit named events and lets other parts of the code listen and react to those events.

```
// Register a listener for the 'greet' event
myEmitter.on('greet', (name) => {
console.log(`Hello, ${name}!`);
});
// Emit the 'greet' event
myEmitter.emit('greet', 'Pikachu');  // Output: Hello, Pikachu!
```

on(eventName, listener) !’ register a listener for an event.

emit(eventName, ...args) !’ trigger the event, passing arguments if needed.

All Node.js streams inherit from EventEmitter, which is why we can listen for events like data, end, error, etc.

data Readable Emitted when a chunk of data is available.

end Readable Emitted when there is no more data to read.

error Both Emitted when an error occurs.

finish Writable Emitted when all data has been flushed to the destination.

close Both Emitted when the stream is closed (may happen after end or finish).

pipe Readable Emitted when the readable stream is piped to a writable stream.

unpipe Readable Emitted when a stream is unpiped from a destination.

```
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 to read.');
});
readableStream.on('error', (err) => {
console.error('Error:', err);
});
readableStream.on('close', () => {
console.log('Stream closed.');
});
```

data !’ called multiple times with chunks of data.
end !’ called once when all data is read.
error !’ called if something goes wrong.
close !’ called when the stream is completely closed.

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

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

writableStream.write('Hello ');
writableStream.write('World!\n');
writableStream.end();
writableStream.on('finish', () => {
console.log('All writes are finished.');
});
writableStream.on('error', (err) => {
console.error('Error writing:', err);
});
```

finish !’ triggers after end() is called and all data is written.
error !’ triggers on any write error.

on(event, listener) Add a listener (persistent)
once(event, listener) Add a listener that triggers only once
off(event, listener) Remove a listener
emit(event, ...args) Trigger the event with optional arguments
listenerCount(event) Count how many listeners are attached to an event

EventEmitter is the backbone of Node.js’s event-driven architecture.
Streams are EventEmitters, which is why we can attach listeners to data, end, error, etc.
You can customize events using EventEmitter in your own code.
Handling error is crucial; unhandled errors can crash the process.

fs Module:

The fs module in Node.js is a core module used to interact with the file system. It allows you to:
Read and write files
Create and delete directories
Monitor changes in files

Work with streams for large files

It is similar to file handling in other languages, but Node.js provides both asynchronous (non-blocking) and synchronous (blocking) APIs.
Non-blocking; uses callbacks or Promises.

Use fs.existsSync or fs.access if you need to check before deleting.

1. Create:

```
const fs = require('fs'); // for callback-based async + sync
const fsp = require('fs/promises'); // for async promise-based
```

fs.writeFile() / fs.writeFileSync()
writeFile replaces the entire contents unless you pass specific flags like 'a' (append).

```
fs.writeFile('data.txt', 'Hello World', (err) => {
if (err) throw err;
console.log('File created or overwritten');
});
```

Promise version:

```
const fsp = require('fs/promises')
await fsp.writeFile('data.txt', 'Hello World');
console.log('File created/overwritten');
```

2. Read:

```
fs.readFile('data.txt', 'utf8', (err, data) => {
if (err) throw err;
console.log(data);
});
const data = await fsp.readFile('data.txt', 'utf8');
console.log(data);
```

3. Append:

Append always adds to the end of the file.

```
fs.appendFile('data.txt', '\nMore text', (err) => {
if (err) throw err;
console.log('Text appended');
});
```

Cannot insert data:
read the file

modify string
write back

```
let content = fs.readFileSync('data.txt', 'utf8');
content = content.slice(0, 5) + "INSERTED" + content.slice(5);
fs.writeFileSync('data.txt', content);
```

4. Delete:

```
fs.unlink('data.txt', (err) => {
if (err) throw err;
console.log('File deleted');
});
await fsp.unlink('data.txt');
```

5. Rename:

```
fs.rename('data.txt', 'data-renamed.txt', (err) => {
if (err) throw err;
console.log('File renamed');
});
await fsp.rename('data.txt', 'data-renamed.txt');
```

File system flag:
'w' write (default), replace file
'a' append
'wx' write only if file does NOT exist
'ax' append only if file does NOT exist

```
fs.writeFile('data.txt', 'Hello', { flag: 'a' }, () => {});
```

Streams process data in chunks (usually 64KB), so memory stays low.

```
const stream = fs.createReadStream('big.txt', { encoding: 'utf8' });

stream.on('error', err => {
console.error('Error:', err);
});

writeStream.on('finish', () => console.log('Write finished'));
writeStream.on('error', err => console.error(err));
```

```
fs.createReadStream('big.txt')
.pipe(fs.createWriteStream('copy.txt'))
.on('finish', () => console.log('Copy done'));

'  Extremely memory efficient
'  Automatically handles backpressure
```

Backpressure Handling:

Node automatically manages backpressure with .pipe().

```
const read = fs.createReadStream('big.txt');
const write = fs.createWriteStream('out.txt');

read.on('data', chunk => {
const ok = write.write(chunk);
if (!ok) {
read.pause();
write.once('drain', () => read.resume());
}
});
```

write.once('drain', ...) fires ONLY when the writable stream’s internal buffer becomes completely empty, not when there is just “one slot” free.

drain = buffer has been fully flushed to the OS, and the stream is ready to accept more data.

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

fs.mkdir('myFolder', (err) => {
if (err) throw err;
console.log('Folder created');
});
```

Create nested folders (recursive: true):

```
fs.mkdir('parent/child/grandchild', { recursive: true }, (err) => {
if (err) throw err;
console.log('Nested folders created');
});
```

List files inside folder:

```
fs.readdir('myFolder', (err, files) => {
if (err) throw err;
console.log('Files:', files);
});
```

Delete an empty folder:

```
fs.rmdir('myFolder', (err) => {
if (err) throw err;
console.log('Folder deleted');
});
```

Delete a folder with files inside (recursive delete):

Child processes allow you to offload CPU-heavy tasks:

Without child processes, horizontal scaling won’t help because:
The request still hits a single event loop
That loop gets blocked
The process becomes unresponsive
Child process = background worker
Does not block main server
Works on separate core

Prevents write buffer overflowing

Horizontal scaling adds processes across machines, not parallel execution inside one app.

Child processes:

Run in true parallel on separate CPU cores
Use multi-core power on the same machine
Communicate via IPC (not network)

Horizontal scaling can scale workers, but a single node still needs a local worker mechanism.

fork():

Sets up IPC channel (parent !” child)
Ideal for offloading CPU-intensive JS tasks

spawn():

Run any system command
Streams output in real-time (stdout, stderr)
Good for long-running processes

exec():

Runs a command in a shell
Buffers all output in memory !’ not good for huge output
Simple for small commands

IPC channel exists only for fork() (Node.js modules)
Communication happens via send and on('message'):

parent.send({ data: 'hello' });
child.on('message', msg => console.log(msg));

For spawn/exec, communication happens via stdout/stderr streams.

Async file, network, database operations are non-blocking
Node uses libuv + OS kernel threads under the hood
Event loop continues while I/O completes
Heavy computation blocks the event loop
Node.js is single-threaded in terms of JS execution. To do CPU-heavy work or run external programs, you need child processes.
Async I/O doesn’t help here, because CPU-intensive loops run inside the main thread
Async I/O doesn’t help here, because CPU-intensive loops run inside the main thread

spawn(command, args):

Starts a new process (any system program)
Run a system command with streaming output
spawn() starts a new process and lets you stream its output in real time.
Used when you want big outputs, or the process is long-running.

const ls = spawn('ls', ['-lh']); // 'ls -lh' on Linux/Mac

```
ls.stdout.on('data', (data) => {
console.log(`stdout: ${data}`);
});
ls.stderr.on('data', (data) => {
console.error(`stderr: ${data}`);
});
ls.on('close', (code) => {
console.log(`child process exited with code ${code}`);
});
```

stdout and stderr are streams !’ you can read output chunk by chunk
The parent process is still non-blocking

exec(command):

Run a command in a shell and get all output buffered
exec() starts a shell (like bash, cmd.exe) and runs your command
Waits until the command finishes
Buffers all stdout and stderr into memory !’ dangerous for huge output

fork(modulePath):

Starts a new Node.js process to run a JS module
Spawn a Node.js module with IPC channel
Child runs a separate Node.js process !’ true parallelism
Ideal for offloading CPU-heavy JS tasks
Avoids blocking main Node event loop

```
const child = fork('worker.js');
child.send({ task: 'compute', value: 42 });
child.on('message', (msg) => {
console.log('Child replied:', msg);
});
```

```
process.on('message', (msg) => {
console.log('Parent sent:', msg);
const result = msg.value * 2;
process.send({ result }); // send back to parent
});
```

process.send() and child.on('message') (built-in IPC)

Async I/O = kernel handles network/file/database operations in background
CPU-heavy JS = cannot be offloaded — Node is single-threaded
Child process solves CPU-bound blocking !’ main thread stays responsive

Global Object in Node.js:

In Node.js, the global object is similar to window in browsers. It provides variables and functions accessible everywhere without requiring require or import.

Node.js automatically provides global.
Variables attached to global become globally accessible.
Best practice: avoid polluting global scope to prevent conflicts.

global.myVar = 'Hello World';

dirname !’ current directory path
filename !’ current file path
console !’ logging
setTimeout, setInterval !’ timers
Buffer !’ binary data handling
process !’ Node.js process info

Process Object

The process object provides information and control over the Node.js process.

process.argv Array of command-line arguments
process.env Environment variables
process.cwd() Current working directory
process.exit([code]) Exit the process with a code
process.pid Process ID
process.on('exit') Event triggered before process exits
process.stdout Writable stream to stdout
process.stdin Readable stream from stdin

The process object in Node.js is a global object that provides information about, and control over, the current Node.js process

The process.argv array contains the command-line arguments passed when starting the Node.js process.

process.env gives access to environment variables.

You can gracefully terminate a Node.js process with process.exit().

Listening for Events: Node.js emits events on the process object.
exit event : Cleanup before shutdown.
uncaughtException event : Handle unexpected errors without crashing the app immediately.

You can directly read/write using process.stdin and process.stdout.
process.stdout.write('Enter your name: ');
process.stdin.on('data', (data) => {});

console.log(process.pid); // process id
console.log(process.platform); // 'darwin', 'win32', 'linux', etc.

You can handle OS signals like SIGINT (Ctrl+C) or SIGTERM.
process.on('SIGINT', () => {
console.log('Caught SIGINT, cleaning up...');
process.exit();
});
Graceful shutdown in servers or daemons.

1. process.argv – Command-line arguments

process.argv is an array of strings containing the command-line arguments passed to the Node.js process.
process.argv[0] !' Node executable path
process.argv[1] !' Path of your script
process.argv[2+] !' Your custom arguments

2. process.env – Environment variables

process.env is an object containing the system environment variables.
process.env.PATH

3. process.cwd() – Current working directory

Returns the directory from where you executed the script, not necessarily where the script is.
Resolving file paths relative to where the user runs the script.
process.cwd() always returns the absolute path of the directory from which the Node.js

process was started, not the folder where the script file (server.js) lives.

cd my-project/server
node server.js

CWD: /full/path/to/my-project/server
dirname: /full/path/to/my-project/server

cd my-project
node server/server.js

CWD: /full/path/to/my-project
dirname: /full/path/to/my-project/server

dirname:

The directory where the script file itself exists
dirname always gives the absolute path of the folder where the current script file physically exists,
dirname is per-file, not per-project.

4. process.exit([code]) – Exit the process

Stops the Node.js process immediately.
code = 0 !’ success
code = 1 !’ failure

5. process.pid – Process ID

Returns the unique process ID assigned by the OS.
console.log('Process ID:', process.pid);

6. process.on('exit') – Event before exit

Event fired just before the process exits. The callback gets the exit code.
Cleanup tasks (closing files, saving logs, releasing resources).

7. process.stdout – Writable stream to stdout

Represents standard output (console), can write text directly. CLI apps or custom output without console.log

process.stdout.write('Hello from stdout\n');

8. process.stdin – Readable stream from stdin

Represents standard input, can read user input from terminal. Interactive CLI apps.

process.stdin.on('data', (data) => {
process.exit();
});

Fork:

fork is a method from the child_process module.
It creates a new Node.js process and establishes an IPC (inter-process communication) channel between the parent and child.
It is specifically designed for running another Node.js script.
Unlike spawn, which can run any command, fork is optimized for Node.js-to-Node.js communication.

To offload CPU-heavy tasks to another process.
To avoid blocking the main event loop.
To communicate easily between parent and child using messages (child.send() / process.on('message')).

const { fork } = require('child_process');

// Fork the child process
const child = fork('./child.js');
// Send data to child
child.send({ numbers: [1, 2, 3, 4, 5] });
// Listen for messages from child
child.on('message', (message) => {
console.log('Parent received:', message);
});
// Listen when child exits
child.on('exit', (code) => {
console.log(`Child exited with code ${code}`);
});

// Listen for message from parent
process.on('message', (msg) => {
console.log('Child received:', msg);

```
const sum = msg.numbers.reduce((a, b) => a + b, 0);
// Send result back to parent
process.send({ sum });
// Optional: exit child process after work
process.exit(0);
});
// Accessing own process info
console.log('Child process PID:', process.pid);
console.log('Child current directory:', process.cwd());
```

process.on('message')
Listens for data from the parent.
Without this, the child cannot receive messages.

process.send()
Sends a message back to the parent through the IPC channel.
Only available in processes created with fork.

process.pid
Shows the child’s process ID, helpful for logging or debugging.

process.cwd()
Shows the current working directory of the child.

process.exit()
Allows the child to exit after finishing its task.

The parent uses fork to offload tasks.
The child uses the process object to communicate and control its own life cycle.

DNS:

DNS (Domain Name System) is like the phonebook of the internet.
Humans access websites using domain names like google.com.
Computers communicate using IP addresses like 142.250.190.78.
DNS translates domain names to IP addresses, allowing your machine to find and connect to servers.

DNS Lookup in Node.js

Node.js provides a dns module to perform DNS queries, allowing your program to resolve

domain names to IP addresses or perform other DNS operations.
If you're using the http or https modules in Node.js to make requests, you need to resolve the domain name into an IP address.

```
dns.lookup('example.com', (err, address, family) => {
if (err) {
console.error('DNS Lookup failed:', err);
return;
}
console.log('IP Address:', address); // e.g., 93.184.216.34
console.log('IP Family:', family);   // 4 for IPv4, 6 for IPv6
});
```

dns.lookup() uses the OS DNS resolver (fast, cached).
Returns one IP address by default.

A single domain can have multiple IP addresses. This is quite common in modern network configurations and is often done for:
Load balancing, Geolocation-based routing, Failover and redundancy

Can specify {all: true} to get all IPs:

```
dns.lookup('example.com', { all: true }, (err, addresses) => {
console.log(addresses);
// Output: [{ address: '93.184.216.34', family: 4 }]
});
```

In Node.js, DNS lookups are done using the operating system's (OS) resolver because it's the most efficient way to perform this task.

Virtual Hosts:

Many servers host multiple domains (virtual hosts) on the same IP. When you access an IP directly, the server might not know which domain you intended to access, so it doesn't respond correctly. This is especially common with shared hosting environments.
Many servers use the same IP address to serve multiple domains. This is done using virtual hosting. In this setup, when a request is made to a single IP, the server looks at the Host header in the HTTP request to determine which domain you are requesting.
The server uses the Host header in the HTTP request to figure out which domain the client is trying to access.

Port Issues: Browsers default to HTTP (port 80) or HTTPS (port 443). If the server is listening on a different port, you need to specify that in the URL (e.g., http://192.168.1.1:8080).

Reverse Proxies and Load Balancers: An IP could point to a load balancer or reverse proxy, which can forward the request to different backend services based on factors like URL path, headers, or even the load on each service. So, even if a request goes to a single IP, that IP might serve requests for multiple domains or services.

The DNS resolver (via the OS resolver or an external service like Google's DNS) queries the DNS records for that domain. The A record (or AAAA for IPv6) typically maps the domain name to an IP address.

If there are multiple IP addresses returned for the domain (because of load balancing or failover), the resolver might pick one at random (or based on other criteria like the closest geographic server) to connect to. If the domain uses a CDN, the DNS query might resolve to an IP address of the nearest edge server, depending on the CDN's routing rules.

dns.lookup()

Acts like the OS-level DNS resolver (like what your computer uses).
It is optimized for hostname !' IP resolution. Default behavior returns first found IP.

dns.resolve()

Performs a true DNS query directly to a DNS server (bypasses OS resolver). Returns all records, not just one.

dns.resolve() allows you to query specific DNS record types, like A, AAAA(ipv6)

```
dns.resolve('example.com', 'A', (err, addresses) => {
if (err) throw err;
console.log('A records:', addresses); // All IPv4 addresses
});
```

Host:

A host is the address of a device on a network.

It can be:

A domain name !' example.com

An IP address !’ 192.168.1.10 (IPv4) or 2001:db8::1 (IPv6)
localhost !’ your own computer (127.0.0.1)
The host tells the network which machine to send data to.

Port:

A port is like an "apartment number" inside the host machine.
It tells the operating system which application should receive the data.

80 !’ HTTP websites
443 !’ HTTPS websites
22 !’ SSH
3306 !’ MySQL

Even if a machine has ONE IP, it has 65,535 ports.
Each application “listens” on a specific port.

Virtual hosting means:
You can host multiple domains on the same IP using Host headers.
The client's browser includes a Host header, The reverse proxy uses that to send the request to the correct backend app.

123.45.67.89
% %siteA.com !’ app on port 3000
% %siteB.com !’ app on port 5000
% %api.siteC.com !’ app on port 8000

-Excluded Virtual Hosting
IP + Port uniquely identifies which app receives traffic.
IP + Port = uniquely identifies ONE application on ONE machine.

The IP 203.0.113.5 might host:
shop.com
blog.com
photos.com

They all share the same IP
BUT they include different Host headers.

Host (domain name) is NOT part of the routing identity.
It is only used after the connection reaches the server, not before.

Because when multiple websites share the same IP + same port, the server uses the Host header to decide which website to serve.

SNI (Server Name Indication): With HTTPS (SSL/TLS), when connecting to the server, you often need to tell the server which site you're trying to access, as the server may host multiple domains on the same IP. This is where SNI (Server Name Indication) comes into play—an extension to SSL/TLS that allows a client to tell the server which hostname it's attempting to connect to.
SNI allows the client (browser) to tell the server which hostname it wants during the TLS handshake.
This way, the server can choose the correct SSL certificate for that hostname. SNI makes HTTPS virtual hosting possible on the same IP and port.

-Virtual Hosting
Virtual hosting allows multiple different websites to run on the same IP + port combination.

The browser sends the Hostname (and for HTTPS, also SNI) inside the request after connecting.

Example:

IP Address: 203.0.113.10
On that IP, one physical machine (or load balancer) hosts:
companyA.com
companyB.com
mysite.net

All use HTTPS on port 443

https://companyA.com ->

DNS Lookup ->

203.0.113.10 ->

Your browser opens a TCP connection: (203.0.113.10:443) "At this point, the server still doesn't know which website you want." ->

For HTTPS, before the encrypted session starts, the browser sends:
SNI = companyA.com
SNI is sent in cleartext, before encryption.

This is THE mechanism that allows many HTTPS sites on the same IP:443.
Based on SNI, the server chooses the correct SSL certificate and the correct virtual host configuration. ->

After TLS is established, browser sends HTTP request. Now inside encrypted HTTPS, the browser sends:

GET / HTTP/1.1
Host: companyA.com

Multiple apps (websites) can live behind the same IP + Port.
Then the app itself decides which internal “site” receives the traffic based on:
Host header (HTTP/1.1)
SNI (HTTPS TLS handshake)
So IP + Port alone is not enough when virtual hosting is used.

```
// Include the `hostname` in the options to use SNI and virtual hosting correctly
const options = {
hostname: 'example.com',
port: 443,
path: '/',                // Path to access
headers: {
'Host': 'example.com'   // This header tells the server which virtual host to serve
}
};

https.get(options, (res) => {
console.log('Status code:', res.statusCode);


dns.lookup('example.com')  // returns IP
https.get(`https://${address}`)
```

Now you're breaking:

SNI (TLS won't know the domain)
Host header (HTTP won't know the domain)

So you must manually add them back:

```
hostname: 'example.com'
headers: { Host: 'example.com' }
```

The browser cannot open a TCP connection using only the domain name.
TCP sockets need: IP address + port
So DNS lookup is needed just to locate the server.

Path Module:

The path module provides utilities for working with file and directory paths in a way that is cross-platform

path.basename(pathString, [ext]):
Returns the last portion of a path (file name).
If ext is provided and matches the file’s extension, it is removed from the result.

path.dirname(pathString):
Returns the directory containing the file.

path.extname(pathString):
Returns the file’s extension, including the leading dot.

path.join(...paths):
Joins path segments together cleanly, inserting separators as needed and removing redundant slashes.
path.join('home', 'user', 'docs');
// !’ 'home/user/docs'

path.resolve(...paths):
Resolves paths into an absolute path.
It works right to left until it finds an absolute path and then resolves from there.
path.resolve('docs', 'file.txt');
// !’ '/absolute/current/working/dir/docs/file.txt'

path.normalize(pathString):
Cleans a path string by resolving .., ., duplicate slashes, etc.

path.parse(pathString):
Breaks a path into its component parts:
root, dir, base, ext, and name.
path.parse('/home/user/docs/file.txt');
/*
{
root: '/',

dir: '/home/user/docs',
base: 'file.txt',
ext: '.txt',
name: 'file'
}
*/

path.format(pathObject):
The reverse of path.parse():
Creates a path string from an object containing parts.
path.format({
dir: '/home/user/docs',
name: 'file',
ext: '.txt'
});
// !' '/home/user/docs/file.txt'

Buffer:

A Buffer is a temporary memory storage for binary data in Node.js.
It allows Node.js to handle raw binary data like files, TCP streams, or network packets.
Node.js uses Buffers because JavaScript strings are UTF-16 encoded, which is inefficient for raw bytes.
Buffer is not part of the standard JavaScript, but Node.js provides it.
Fixed size, cannot be resized after creation (like an array of bytes).

const buf = Buffer.alloc(10); // Creates buffer of 10 bytes, filled with 0
console.log(buf); // <Buffer 00 00 00 00 00 00 00 00 00 00>

const buf3 = Buffer.from([1, 2, 3, 4]);
console.log(buf3); // <Buffer 01 02 03 04>
const buf4 = Buffer.from('Hello');
console.log(buf4); // <Buffer 48 65 6c 6c 6f>

Event Loop:

Node.js is single-threaded, but can handle many concurrent operations using an event-driven, non-blocking I/O model.
The event loop is the mechanism that checks for and executes callbacks from asynchronous operations.
Node.js constantly checks what tasks are ready and executes them in order.

Phases of the Event Loop:

1. Timers
2. Pending Callbacks
3. Idle, Prepare (internal use)
4. Poll
5. Check
6. Close Callbacks

Additionally, microtasks like process.nextTick() and Promises have their own queues.

Phase 0: Microtasks

Microtasks run before the next phase of the event loop.
Includes:
process.nextTick()
Promises (.then, .catch, .finally)

```
Promise.resolve().then(() => console.log('promise'));
process.nextTick(() => console.log('nextTick'));
```

Phase 1: Timers

Executes callbacks scheduled by setTimeout and setInterval whose time has elapsed.

Phase 2: Pending Callbacks

Executes I/O callbacks that were deferred to the next loop iteration, e.g., some TCP errors, fs errors.
These are NOT regular async callbacks.
This phase is only for special internal callbacks that need to be executed before the poll phase can continue.

Phase 3: Poll

Retrieves new I/O events from the OS.
Executes I/O callbacks (e.g., reading from disk, network).
If no I/O, Node.js may wait here for new events or move to the next phase.
Most I/O callbacks, like HTTP requests or file reads, run in this phase.
Most I/O callbacks, like HTTP requests or file reads, run in this phase.
fs.readFile

fs.writeFile

Phase 4: Check

Executes setImmediate() callbacks.
setImmediate always runs in the check phase, after poll phase finishes.

Phase 5: Close Callbacks

Handles close events, e.g., socket.on('close', ...) or fs.close() callbacks.

Inside an I/O callback, the order is guaranteed:
setImmediate !’ setTimeout(0)

On startup, Node runs:
Synchronous code
Enters event loop starting at the POLL phase, not Timers.

EJS:

EJS (Embedded JavaScript) lets you render HTML templates dynamically.

HTTP Protocol

HTTP (Hypertext Transfer Protocol) is the foundation of web communication between clients (browsers, apps) and servers.
It is an application-layer protocol (runs on top of TCP/IP).
Request-response model:
Client sends a request: GET /index.html HTTP/1.1
Server responds with status code, headers, and content.
Stateless: Each request is independent; server does not store previous client info by default.

SSL (Secure Sockets Layer):

Early protocol for encrypting communication between client and server.
Secures data to prevent eavesdropping or tampering.

TLS (Transport Layer Security)

Modern, secure replacement for SSL.
Encrypts HTTP traffic !’ HTTPS.

TLS Handshake Workflow

Client connects to server !’ requests secure connection.
Server sends digital certificate (proves identity).
Client validates certificate (issued by trusted Certificate Authority).
Both agree on encryption keys.
Encrypted communication begins.

HTTPS = HTTP + TLS/SSL !’ secure version of HTTP.
Browser shows lock icon when site uses HTTPS.
Prevents MITM (Man-In-The-Middle) attacks and data leakage.

Certificate:
A digital file issued by a Certificate Authority (CA).
Confirms server identity.

Includes:

Server public key
Domain name
Expiration date
CA signature

HTTP is a protocol—a set of rules describing how a client (browser/app) talks to a server.

HTTP/1.0:
One connection per request.
No persistent connections by default
No Host header (broke when multiple sites shared a server)
It treated every request independently, meaning the browser had to open a new TCP connection for each file—HTML, images, CSS, JavaScript—making webpages slow and inefficient. It had no built-in support for keeping connections alive or hosting multiple websites on one server.

HTTP/1.1:
Persistent connections (Keep-Alive), One TCP connection handles multiple requests.
Server can send data in pieces — great for streaming.
It introduced persistent connections, so multiple requests could reuse the same TCP connection instead of opening a new one each time. It also added better caching rules, chunked responses (for streaming content), and the Host header so multiple domains could live on the same server.

HTTP/2:

Multiplexing, Many requests/responses interleaved over a single connection.
Binary framing, Data is sent in binary frames !’ faster parsing.
It introduced multiplexing, which lets many requests travel simultaneously over a single TCP connection without waiting for each other. It also uses binary framing for efficiency and compresses headers to reduce overhead.

HTTP/3:

Built on top of QUIC (replaces TCP).
QUIC runs over UDP and handles reliability itself.
TCP suffers from head-of-line blocking even with HTTP/2 multiplexing, because TCP delivers packets in strict order.
Replacing TCP with QUIC, a modern protocol built on top of UDP. HTTP/3 avoids TCP’s head-of-line blocking entirely, meaning that a lost packet no longer slows down every request.

QUIC: Quick UDP Internet COnnection

It replaces the traditional TCP layer beneath HTTP and runs on top of UDP, giving it far more flexibility. QUIC was designed to solve long-standing problems in TCP—especially slow handshakes, head-of-line blocking, and poor performance on unstable networks. It delivers built-in encryption, faster connection setup, true multiplexing, and better switching between networks (like moving from WiFi to mobile data) without dropping the connection. Because it removes TCP’s rigid behavior and uses smarter mechanisms at the transport layer, QUIC is the foundation for HTTP/3, the newest and fastest version of HTTP.