# MEA - Makers a new approach to Electronic Applications

Node

Riccardo Berta

2025.11.12

## Contents

# 1 NodeJS

Node.js is a powerful way to **run JavaScript code outside of the web browser**. Originally released in 2009, it opened up new possibilities for JavaScript, allowing developers to build server-side applications using a language that was once confined to the client side. At the heart of Node.js is the V8 engine, the same high-performance JavaScript engine that powers Google Chrome. V8 is built with cutting-edge compiler technologies that make it incredibly fast. In fact, programs written in JavaScript and run with Node.js can achieve performance that rivals C programs, but with a significantly lower development cost and complexity. You can explore more about V8 here: https://v8.dev.

Node.js isn't just a JavaScript engine though. It **extends JavaScript with additional APIs** to handle things like the file system, HTTP requests, and binary data—features you wouldn't normally find in browser-based JavaScript.

One of Node's key features is its **event-driven architecture**. This design allows Node.js to handle concurrency through an event loop, rather than relying on traditional multithreading. This means developers don't need to worry about threads and locks, Node.js handles asynchronous operations behind the scenes in a very efficient way. For example, instead of waiting for a file to load from disk (which might take some time), Node lets the program continue running and calls back a function once the file is ready, just like how browser code listens for an ”onclick” event. This **non-blocking, asynchronous approach** makes Node.js especially well-suited for applications that need to deal with lots of simultaneous requests, like web servers and real-time apps. The programming model is both scalable and intuitive once we understand the event loop.

Node.js can function as **a full-featured web server**, but it's also a **general-purpose runtime environment**. It includes a REPL (Read-Eval-Print Loop), which is an interactive shell where we can write and test JavaScript code line-by-line. This makes it a great environment not only for building production-ready applications but also for learning and experimenting with JavaScript in real-time.

One of the reasons Node.js has gained so much popularity is its ability to simplify the development of scalable network applications. Traditional server technologies often struggle under the weight of concurrent connections, requiring complex infrastructure and significant hardware. Node.js takes a different approach, using its event-driven, non-blocking architecture to handle a large number of simultaneous connections efficiently, without the need for heavy threading or complex concurrency logic. Major companies have experienced dramatic improvements after switching to Node.js. For instance, LinkedIn migrated its backend from Ruby on Rails to Node.js in 2012. The results were remarkable: server usage was reduced to a factor of 10, and the application ran up to 20 times faster. The switch also enabled the frontend and backend teams to collaborate more easily, using the same language across the stack. PayPal made a similar move in 2013, transitioning from Java to Node.js. Their development cycle sped up, with teams building features almost twice as fast and with fewer people involved. Performance improved significantly as well—requests per second doubled, and the average response time dropped by 35%.

## 1.1 Getting Started

To begin using Node.js, head to the official site. Installation is straightforward on most systems. Once installed, you can launch the interactive shell—called the REPL (Read-Eval-Print Loop)—just by typing ”node” in the terminal. This environment is great for experimenting and learning, but it's not intended for running full applications in production. It does offer a few useful meta-commands such as ”.help”, ”.exit”, and ”.clear”, which can assist while testing small snippets.

### 1.1.1 Writing applications

To write real applications, use any code editor to create a JavaScript file, such as ”app.js”, and then run it with the command:

```
node app.js
```

This approach allows you to build structured applications using external libraries and modules, not just quick tests. For development, there are many good options in terms of editors and IDEs. **Visual Studio Code** is a lightweight, powerful editor with great Node.js support and extensions. Other options include IntelliJ IDEA (with its Node.js plugin) and Atom, which can be extended with packages like ”atom-debugger” for Node.js debugging support.

### 1.1.2 NPM: Node Package Manager

One of the biggest advantages of using Node.js is its massive ecosystem of reusable code, made easily accessible through a tool called **NPM**, short for **Node Package Manager**. NPM comes automatically installed with Node.js and serves as both a package manager and a public registry for Node modules. The official site hosts **hundreds of thousands of free packages**. These packages range from utilities for handling dates and strings to powerful web frameworks and database connectors. If there's a problem you're trying to solve in your Node.js app, chances are someone has already built a package for it.

### 1.1.3 Modules

In Node.js, a **module** is simply a reusable piece of code, like a library in other programming languages. Modules help keep code organized and modular by allowing us to separate functionality into files and folders. Node.js includes a set of **core modules** that are available without any installation. These include modules for basic tasks such as:

- `http` – to create web servers
- `url` – for URL parsing
- `querystring` – to parse query parameters
- `path` – for file path utilities
- `fs` – to interact with the file system
- `util` – for utility functions

To install an external moduel, we use the ”npm install” command. For example, to install the ”upper-case” module, just run:

```
npm install upper-case
```

This will create a directory called ”node_modules” in our project folder, where the package (and any dependencies it needs) will be stored. Once the package is installed, using it in our code is simple:

```
let uc = require('upper-case');
console.log(uc.upperCase("hello world")); // Output: HELLO WORLD
```

We can also create our own **local modules** to encapsulate and reuse functionality within our project. This is done using the ”exports” keyword. Here’s a basic example:

```
// myfirstmodule.js

exports.myDateTime = function () {
  return Date();
};
```

Now we can use this custom module in any Node.js file like so:

```
let dt = require('./myfirstmodule');

console.log("Current date and time: " + dt.myDateTime());
```

Notice that when loading a local file module, we must use "./" to indicate that it's in the current directory.

### 1.1.4 Manifest

Every Node.js project is centered around a special file called "package.json". Think of this file as the "manifest" of our project: it stores essential metadata that identifies the project and outlines how it should behave, what dependencies it needs, and how it can be run or used by others.

The package.json includes **descriptive metadata** such as:

- The project's name
- The current version
- A short description
- The author and license

It also includes **functional metadata** that tells Node.js how to run the app:

- The entry point file (like "index.js")
- A list of dependencies (other modules our code needs)
- Links to source code repositories or bugs reporting tools

To get started, create a new folder for our project and run:

```
npm init
```

This command will ask us a series of questions (like the project name, version, entry file, etc.) and then generate a "package.json" file based on our answers:

```
{
  "name": "test",
  "version": "0.0.1",
  "description": "This is an example",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Riccardo Berta",
  "license": "MIT"
}
```

We can then add a dependency to our project, such as:

```
npm install upper-case
```

This installs the ”upper-case” module and automatically adds it to the ”dependencies” section of package.json:

```
{
    "name": "test",
    "version": "0.0.1",
    "description": "This is an example",
    "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Riccardo Berta",
    "license": "MIT",
    "dependencies": {
      "upper-case": "^2.0.2"
    }
}
```

In package.json, dependencies are categorized into two main types:

- **dependencies**: these are packages our app needs to run in production. For example, an HTTP server or a database client.
- **devDependencies**: these are packages used during development only, like testing tools, linters, or bundlers. They aren't needed in production.

When distributing our Node.js project, we typically **exclude** the ”node_modules” folder (since it can be huge) and rely on the package.json file instead. Anyone who clones our project can simply run:

```
npm install
```

This command installs all the modules listed under "dependencies" in the package.json file.

### 1.1.5 Version Control

**Semantic Versioning** is a system that uses a three-part version number to convey meaning about the changes in a package. This helps developers manage dependencies in a predictable and reliable way. A version number looks like this:

```
MAJOR.MINOR.PATCH
```

For example, in 1.2.3:

- 1 is the **major** version
- 2 is the **minor** version
- 3 is the **patch** version

A **Major** changes (1.x.x -> 2.0.0) introduce **breaking changes** that may not be backward compatible. A **Minor** changes (1.2.x -> 1.3.0) add new features but **maintain compatibility** with older versions. A **Patch** changes (1.2.3 -> 1.2.4) are **bug fixes or small improvements** that don't affect compatibility.

When we install a package with NPM, the version is recorded in package.json and to provide flexibility in updates, we can use symbols to define **version ranges**:

- ~1.2.3 means to install 1.2.3 or any later **patch** version, like 1.2.4, 1.2.5, etc, but not 1.3.0. We use this when we want to accept only safe bug fixes, not new features.
- 1̂.2.3 means to install 1.2.3 or any **minor** or **patch** updates, like 1.3.0, 1.4.1, etc., but not 2.0.0. We use this when we are OK with new features that don't break compatibility. Be cautious if the library's authors don't strictly follow semantic versioning.
- * means to accept **any version**, regardless of major, minor, or patch. We are telling NPM to use whatever is the latest version available. This is risky and **not recommended** for production use.
- >=1.2.3 and >1.2.3 means any version **higher/strictly higher** than 1.2.3. It is useful when we want a minimum version (e.g., for a feature introduced in 1.2.3).
- <=1.2.3 and <=1.2.3 means any version **lower/strictly lower** than 1.2.3. It is useful when we want to avoid breaking changes introduced in newer versions.

Consider the following example:

```
"dependencies": {
  "express": "^4.17.1",
  "mongoose": "~6.0.13"
}
```

This means:

- Accept **any 4.x.x** version of Express package starting from 4.17.1
- Accept **only 6.0.x** versions of Mongoose starting from 6.0.13

## 1.2 Synchronous vs Asynchronous

Events are a foundational aspect of the language and its runtime model. From the beginning, JavaScript has been designed with interactivity in mind, especially for browser environments where user interaction—such as clicks, keyboard input, or mouse movements—drives much of the application behavior. However, events are not limited to user-driven interactions. In server-side environments, a much broader and more diverse range of events come into play. These include network requests, database operations, file system access, timers, and many other asynchronous activities that can occur independently of any user input. Understanding how code execution **flows** is crucial to mastering Node.js. JavaScript, especially in the Node.js environment, relies heavily on **asynchronous programming** to manage tasks efficiently without blocking the system. Let's explore what that really means by comparing three common models.

### 1.2.1 Synchronous, Single Thread

In a traditional synchronous, single-threaded execution model, every instruction in a program is carried out in a strict, **sequential order**. This means that the program handles one task at a time, and it cannot move on to the next task until the current one has been fully completed. There is no parallelism or overlapping of operations, each task waits its turn in the line of execution.

For instance, if a program needs to read two files from disk one after the other, it will begin by reading the first file. The second file read operation will not even start until the first one has completely finished. This approach becomes problematic when **any single operation is slow or involves waiting**, such as accessing the file system, performing a network request, or querying a database. During this waiting period, the entire program is effectively paused, unable to do anything else. As a result, the total time it takes to complete all tasks is simply **the sum of the durations of each task**, including the time spent waiting.

Consider the following JavaScript example using synchronous file reading:

```
let data1 = readFileSync("file1.txt"); // blocks the thread until this is done
let data2 = readFileSync("file2.txt"); // only starts after the first one completes
```

In this code, the second file is not read until the first one is entirely loaded. While this model is easy to understand and debug, especially for beginners, it does not perform well in environments where responsiveness and scalability are important. Applications that need to handle many simultaneous tasks, such as web servers or user interfaces, can become sluggish or unresponsive if built entirely on synchronous, single-threaded logic.

### 1.2.2 Synchronous, Multiple Threads

To enhance the ability of a system to perform multiple tasks concurrently, many programming environments and operating systems make use of multiple threads of execution. In this model, instead of processing every task sequentially within a single thread, **the system creates and manages multiple threads**, each capable of executing a different task. These threads can run simultaneously, leveraging the processing power of multi-core CPUs. When a program follows this approach, tasks that would otherwise block the main thread (such as computationally intensive operations or input/output activities) can be offloaded to separate threads. These threads operate independently, and they may begin **executing in parallel**, without having to wait for the completion of previous tasks. This greatly improves responsiveness and throughput, especially in applications that serve many users or perform many operations at once.

However, because these threads share the same memory space, **coordination between them becomes crucial**. Often, the main thread must eventually pause to wait for the completion of other threads, collect their results, and proceed only after re-establishing a consistent state. This process is known as **synchronization**. Synchronization is essential to ensure that shared resources are accessed in a safe and predictable way, but it also introduces complexity. Developers must be cautious of common pitfalls such as race conditions, deadlocks, and other **concurrency bugs** that are difficult to detect and reproduce.

This model is widely used in languages such as Java and C++, particularly in systems that require high performance under concurrent loads. Examples include multithreaded web servers that handle multiple client requests at once, or desktop applications that maintain a responsive user interface while performing background computations. While powerful, the multithreaded model demands careful design and discipline to avoid the subtle and often critical issues that arise from improper thread handling.

### 1.2.3 Asynchronous Execution Model

Unlike traditional synchronous models that rely on sequential or multi-threaded execution, Node.js adopts an entirely different strategy: it is built around **asynchronous programming** using an **event-driven architecture**. This means that instead of blocking the execution flow while waiting for slow operations, like file access or network communication, to complete, the program initiates these operations and then immediately moves on to the next line of code.

In practice, when Node.js encounters a time-consuming task (reading a file from disk) it begins that operation and immediately continues executing the rest of the program. Rather than stalling the entire thread until the result is ready, Node.js registers a **callback function** (or, in modern syntax, attaches a **promise**) that will be invoked once the operation completes. When the system finishes the requested task, it executes the callback. For example, consider the following code using Node.js's asynchronous file reading capability:

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

fs.readFile("file.txt", (err, data) ⇒ {
  if (err) throw err;
  console.log(data.toString());
});

// Meanwhile, the app can continue doing other things
```

In this code, the fs.readFile() function initiates the file read, but it does not wait for it to finish before moving forward. Instead, the rest of the application remains free to perform other work, such as responding to additional requests or executing unrelated code. When the file has been fully read, the callback function is triggered, and the data is processed.

This asynchronous, non-blocking model is a cornerstone of Node.js and is the reason it can handle thousands of simultaneous I/O operations efficiently, all within a single thread. There is no need to spawn new threads or manage complex synchronization mechanisms, as the callback system is designed to manage concurrency in a lightweight and elegant way.

This model is especially effective in scenarios where applications need to remain highly responsive while performing many slow, I/O-bound tasks, such as serving dynamic web content, handling API requests, or managing large volumes of data coming from databases or network sources. While it may introduce a steeper learning curve compared to straightforward synchronous code, especially in terms of flow control and error handling, the performance benefits and scalability offered by asynchronous programming make it an ideal choice for modern server-side applications.

### 1.2.4 Callback hell

While the callback approach worked, it often led to a situation known as **callback hell**, where multiple nested callbacks created code that was difficult to read, maintain, and debug. This pattern quickly became problematic as the complexity of logic increased. Consider the following example:

```
getUser(userId, function(err, user) {
  if (err) {
    console.error('Error fetching user');
  } else {
    getOrders(user, function(err, orders) {
      if (err) {
        console.error('Error fetching orders');
      } else {
        getOrderDetails(orders[0], function(err, details) {
          if (err) {
            console.error('Error fetching order details');
          } else {
            console.log('Order details:', details);
          }
        });
      }
    });
  }
});
```

Each function is nested within the previous one, creating deeply indented and hard-to-follow code. This is difficult to manage, especially when the nesting gets deeper or we need to add error handling at multiple levels.

### 1.2.5 Function Chaining

Function chaining addresses the deep nesting issue by allowing asynchronous operations to be linked in a linear, readable sequence. This is made possible with **Promises**, a core feature in modern JavaScript that represents the eventual completion or failure of an asynchronous task. Promises support methods like .then() and .catch(), which are specifically designed to be chained.

When a function returns a Promise, you can call .then() on it to specify what should happen once the Promise is fulfilled. The .then() method itself returns a new Promise, which allows you to chain another .then(), creating a smooth, left-to-right flow of asynchronous logic. If any step in the chain fails, the error can be caught and handled with a .catch() block at the end of the chain, without disrupting the rest of the code. For example, instead of nesting callbacks like:

```
getUser(userId, function(err, user) {
    getOrders(user, function(err, orders) {
      getOrderDetails(orders[0], function(err, details) {
        console.log(details);
      });
    });
  });
```

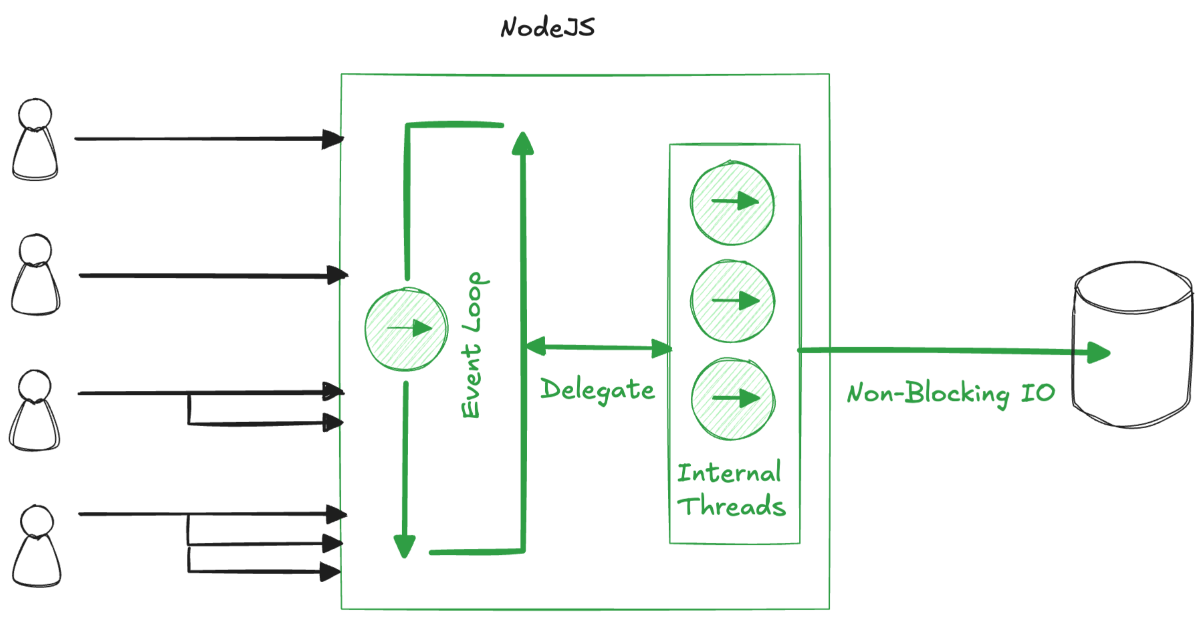
We can refactor the same logic using function chaining and Promises:

```
getUser(userId)
  .then(user ⇒ getOrders(user))
  .then(orders ⇒ getOrderDetails(orders[0]))
  .then(details ⇒ console.log(details))
  .catch(error ⇒ console.error('Something went wrong:', error));
```

This structure is much easier to follow and scales better as more steps are added. Each function in the chain returns a Promise, and the next .then() waits for that Promise to resolve before continuing. Errors from any point in the chain can be caught and handled in one centralized place using .catch(). Function chaining, enabled by Promises, is a major improvement in JavaScript's approach to asynchronous programming, offering a cleaner, more manageable alternative to nested callbacks.

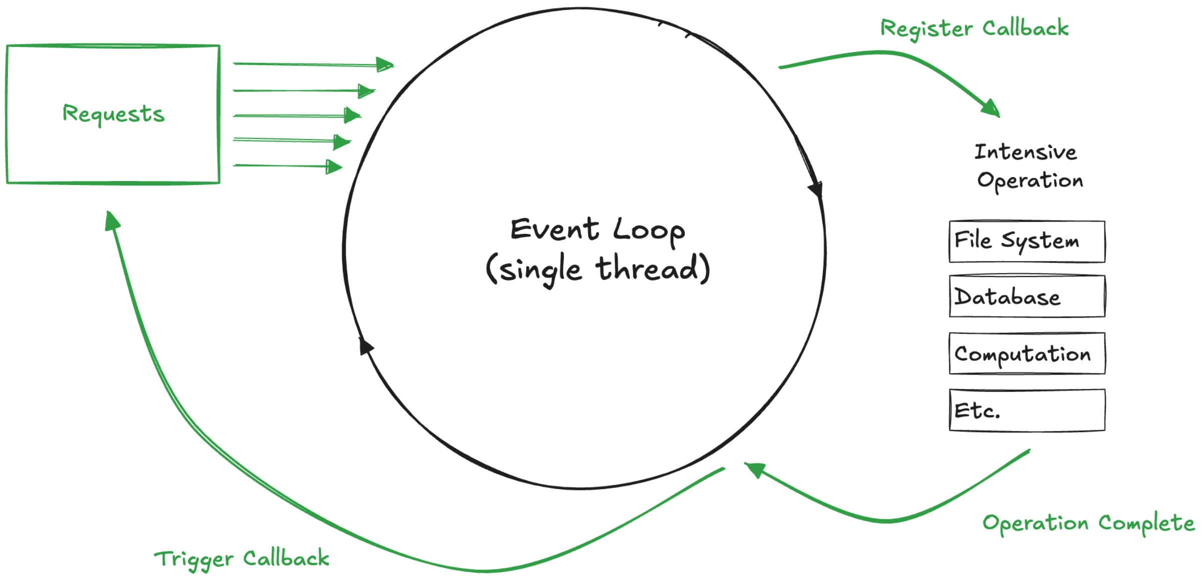
### 1.2.6 Event Loop

To fully understand the asynchronous model adopted by Node.js, we need to explore a key concept at the heart of its runtime: the **Event Loop**. JavaScript is a single-threaded language (it can execute only one instruction at a time) but thanks to the Event Loop, it can handle asynchronous events in a highly efficient, non-blocking way. When a JavaScript program runs in Node.js, there is a single main thread dedicated to executing JavaScript code. This thread is responsible for handling all incoming events (from HTTP requests and timers to file system operations and more). As the program runs, time-consuming tasks (like file access or database queries) are **delegated to the system's underlying non-blocking I/O mechanisms**, such as the pool of internal threads.



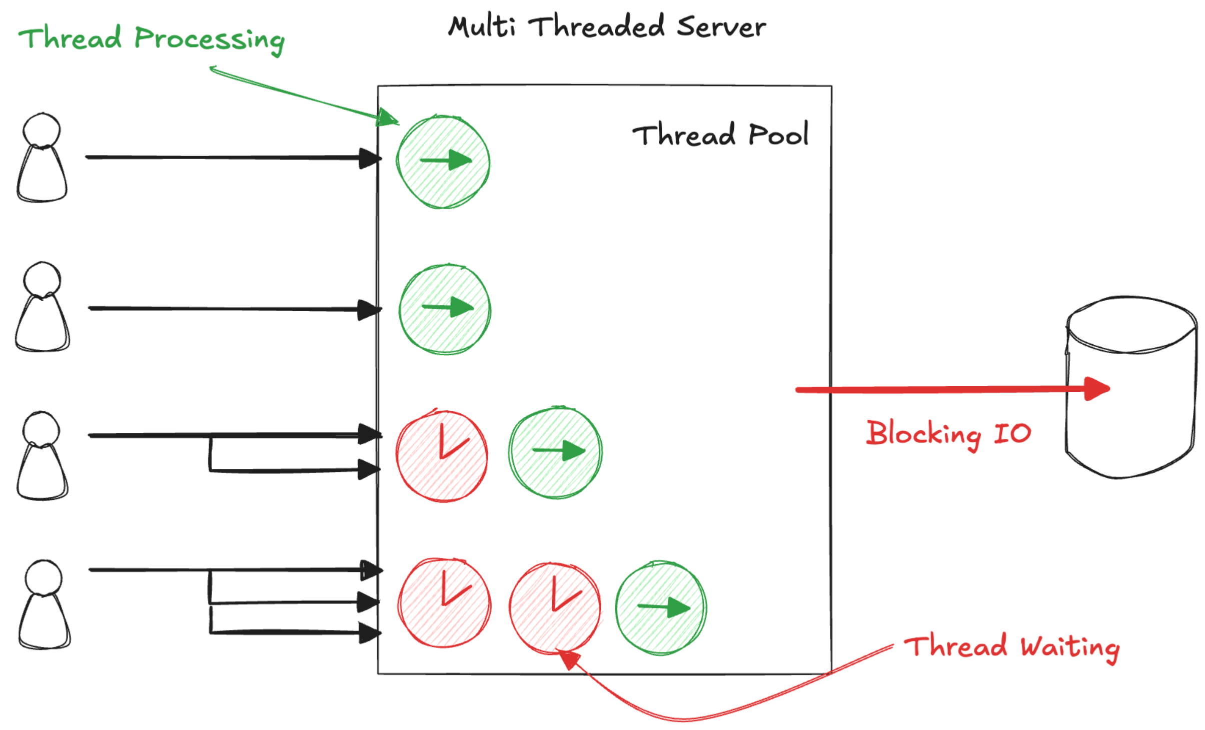
These tasks do not block the main thread. Instead, once the task completes, the system emits an event which gets placed in a **task queue**. The Event Loop constantly monitors this queue, and when the main thread is idle, it picks up the next task and executes its associated callback:

```
while(queue.waitForMessage()){
    queue.processNextMessage();
}
```



Here, multiple requests arrive and are quickly handed off by the single-threaded Event Loop to the non-blocking I/O system. The actual file or database access happens in the background, managed by internal threads. Once the data is ready, the callback is queued back to the main thread, which processes it when it's ready. This allows the Node.js server to continue handling other requests without being held up by slow I/O operations.

This approach contrasts sharply with the traditional **multi-threaded server** model:



In that model, every incoming request is handled by a separate thread from a thread pool. If a request involves a blocking I/O operation (like reading from disk), the thread must wait: it is effectively paused until the operation completes. This leads to a lot of "thread waiting" time, represented by the red clock icons in the figure, which consumes system resources and limits scalability.

The Node.js model minimizes the number of threads that are left idle. Since the Event Loop does not block while waiting for I/O operations to complete, it can keep the single main thread available for processing many more incoming events, making Node.js particularly well-suited for **I/O-heavy** applications like web servers, chat apps, or real-time data services.

This efficiency comes at the cost of complexity: developers must write code that **never blocks the main thread**, and long computations must be offloaded to worker threads or broken into smaller chunks. But when used appropriately, the Event Loop model offers significant performance and scalability advantages without the pitfalls of thread synchronization, race conditions, or deadlocks typical in multi-threaded environments.

## 1.3 The HTTP Module

In Node.js, the http module allows us to create a fully functioning **HTTP server**. Unlike languages like PHP, Java, or C#, which are typically executed inside a separate web server such as Apache or IIS, Node.js can **act as a standalone server itself**. This means it can handle HTTP requests directly without the need for an external server environment. For example, the following code (`01.HelloWorldServer.js`) demonstrates a minimal HTTP server in Node (see 01.HelloWorldServer.js code):

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

http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Hello World\n');
}).listen(8124, "127.0.0.1");

console.log('Server running at http://127.0.0.1:8124/');
```

Here, we import the http module and assign its functionalities to the http object. The createServer() method takes a callback function that is triggered every time the server receives a new request. This function accepts two parameters: request, which contains details such as the URL and headers, and response, which is used to send data back to the client. Finally, the server listens on port 8124 of the localhost.

This demonstrates how Node.js handles events and callbacks. Whenever a request is received (an event), the callback function we've passed is executed. In just six lines, we've created a basic but complete HTTP server. Of course, real-world web servers often need to do more. They typically examine the request method (e.g., GET, POST) to determine the client's intended action, and they parse the requested URL to identify which resource that action applies to.

Node.js can also act as a client to send HTTP requests. The following example (see 02.HelloWorldClient.js code) shows how to make a simple GET request to download an image:

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

var request = http.request({
    hostname: "eloquentjavascript.net",
    path: "/img/cover.jpg",
    method: "GET",
    headers: {Accept: "text/html"}
    },
    function(response) {
        console.log("Server responded with status code: ", response.statusCode);
    });

request.end();
```

In this example, we use http.request() to configure the request: we specify the target server, the path, the HTTP method, and the headers. When a response is received, a callback is triggered to handle it. In this case, logging the status code to the console.

To perform secure requests over HTTPS, Node.js provides a separate https module with a similar interface, including its own request() method.

### 1.3.1 Streams

In many real-world applications, data is not always available all at once. For example, when processing files, receiving input over a network, or handling real-time data feeds, the total size of the data might be large or even unknown at the start. Loading everything into memory at once can be inefficient or even infeasible. To address this, Node.js provides an elegant and efficient solution through **streams**, an abstraction for handling continuous, chunked data.

A **stream** is essentially a sequence of data made available over time. Instead of waiting for the entire dataset to be received or loaded before processing, streams allow you to handle the data in **smaller pieces** (called **chunks**) as it arrives. This streaming approach becomes especially important when working with **large datasets** or **external sources** like file systems, network requests, or stdin/stdout, where the total size or duration of the data flow is not known in advance.

We already encountered streams in earlier examples: the response object returned by the HTTP server and the request object used in http.request() are both instances of streams. These are **writable** and **readable** streams, respectively. Node.js stream objects provide a consistent interface for interacting with different types of streaming data. There are four fundamental types of streams in Node.js:

- **Readable** streams: represent sources you can read data from (e.g., file input, HTTP requests).
- **Writable** streams: represent destinations you can write data to (e.g., file output, HTTP responses).
- **Duplex** streams: are both readable and writable (e.g., network sockets).
- **Transform** streams: are duplex streams that can modify or transform the data as it is read and written (e.g., compression or encryption streams).

Each stream is an instance of an **EventEmitter** and emits various events such as data, end, error, and close. For example, to manually transfer data from a readable stream to a writable one, we can do:

```
readable.on('data', (chunk) => {
    writable.write(chunk);
  });

readable.on('end', () => {
    writable.end();
  });
```

### 1.3.2 Pipe

While this pattern works, Node.js provides a simpler and more elegant approach through the .pipe() method. **Piping** automatically manages the flow of data from a readable stream into a writable one, handling backpressure and cleanup internally. For instance:

```
readableSrc.pipe(writableDest);
```

This line effectively sets up a connection where readableSrc continuously pushes data into writableDest as it becomes available. The .pipe() method greatly simplifies code when dealing with file transfers, proxies, or any situation where data flows from one source to a destination.

Streams provide the building blocks for efficient, modular, and scalable data processing in Node.js applications. They are the backbone of many performance-sensitive operations, allowing developers to write programs that work with data incrementally and without unnecessary memory consumption.

### 1.3.3 Serving a big file

To fully appreciate the power of streams, let's walk through a practical example that demonstrates the dramatic difference they make in terms of **memory consumption** and **efficiency**. Start by generating a large file (see \03.Stream\create.js code):

```
const fs = require('fs');
const file = fs.createWriteStream('./big.file');
for(let i=0; i<= 1e7; i++) {
    file.write('Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
        ↪  eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
        ↪  veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
        ↪  commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit
        ↪  esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
        ↪  cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est
        ↪  laborum.\n');
}
file.end();
```

This code produces a file named ”big.file”, approximately 400 MB in size. We can experiment serving this large file using a method that **does not use streams** (see \03.Stream\server_1.js)

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

const server = require('http').createServer();

server.on('request', (req, res) ⇒ {
  fs.readFile('./big.file', (err, data) ⇒ {
    if (err) throw err;
    res.end(data);
  });
});

server.listen(8000);
```

A request handler reads the entire big.file into memory using fs.readFile(). Only after the full file is loaded is it written to the response. This approach might seem straightforward, but it is **highly inefficient** for large files. When tested, the server started with about 8.7 MB of memory usage. Upon receiving a single request, it jumped to 434.8 MB, meaning the entire file content had to be loaded into memory at once before being sent out. This kind of **memory spike** may be acceptable for small files, but it becomes **untenable for very large files** or high-traffic environments. In fact, trying the same approach with a 2GB file will likely cause the server to run out of memory or become unresponsive. In contrast, we can use a method based on streams (see \03.Stream\server_2.js):

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

const server = require('http').createServer();

```

```
server.on('request', (req, res) => {
        res.writeHead(200, {'Content-Type': 'text/plain'});
        const stream  = fs.createReadStream('./big.file');
        stream.pipe(res);
});

server.listen(8000);
```

It uses the fs.createReadStream() method to serve the same file. Since the HTTP response object is a **writable stream**, we can directly **pipe** the file stream into the response. This setup ensures that **only small chunks of the file** are loaded into memory at any given time. The system reads from the file and writes to the response stream **incrementally**, keeping memory usage low and stable. In this case, even when handling the same 400 MB file, memory usage only rose by about 25 MB, a massive improvement in efficiency.

Now imagine running both versions of the server with a **2GB file**. With server_1, the process might crash or become unresponsive, as it tries to load the entire 2GB into memory. But with server.js and the stream-based approach, the memory footprint remains minimal, and the server continues to function smoothly, even under load.

This example clearly illustrates the importance of using streams for I/O-heavy operations. Not only do they prevent unnecessary memory consumption, but they also make our application more scalable and responsive under pressure.

## 1.4 Examples

To consolidate the concepts learned so far (including asynchronous programming, streams, and event-driven architecture) we now explore two practical and illustrative Node.js applications: a **File Server** and a **Chat Server**.

These examples serve different purposes but are built on the same foundational principles of the Node.js runtime: efficient handling of concurrent I/O using **non-blocking code** and **event-based logic**. They provide a natural transition from working with the file system and HTTP to managing lower-level TCP socket communication.

- The **File Server** combines HTTP and the file system into a simple interface. It allows remote clients to read, write, and delete files over HTTP using standard methods like GET, POST, and DELETE. By leveraging **streams**, it avoids reading entire files into memory, ensuring efficiency even with very large files.

- The **Chat Server** drops below the HTTP layer and works directly with TCP sockets, enabling real-time communication between multiple clients. It shows how to use Node's **net module** to handle raw socket connections, broadcast messages, and manage disconnections and errors gracefully.

Both projects demonstrate how Node.js scales well without requiring threads, and how its **single-threaded event loop** can effectively manage multiple concurrent connections with minimal resource usage.

These examples are also excellent starting points for building more advanced applications — such as APIs, cloud storage interfaces, multiplayer games, or messaging systems — all while staying true to the Node.js philosophy of lightweight, asynchronous design.

### 1.4.1 A file Server

We create a simple **file server** using Node.js. This server combines everything we've learned (working with the file system, handling HTTP requests, and streaming data efficiently) to allow remote clients to **read, write, and delete files** over the network (see 04.FileServer.js code).

```
const http = require("http");
const fs = require("fs");
const mime = require("mime");

// An object called methods to store the functions that handle
// the various HTTP methods
const methods = Object.create(null);

http.createServer(function(request, response) {

  // A function to finish the request. It takes an HTTP
  // status code, a body, and optionally a content type as arguments.
  // If the value passed as the body is a readable stream,
  // it will have a pipe method, which is used to forward a readable
  // stream to a writable stream. If not, it is assumed to be either
  // null (no body) or a string and is passed directly
  // to the response's end method.
  function respond(code, body, type){
        if (!type) type = "text/plain";
        response.writeHead(code, {"Content-Type":type});
    if (body && body.pipe) body.pipe(response);
    else response.end(body);
  }
  // The respond function is passed to the functions that
  // handle the various methods and acts as a callback to finish
  // the request.
  if(request.method in methods) methods[request.method](urlToPath(request.url),
   ↪  respond, request);
  // Returns 405 error responses, which is the
  // code used to indicate that a given method isn’t handled
  // by the server
  else respond(405, "Method " + request.method + " not allowed.");

}).listen(8000);

// A function to get a path from the URL in the request,
// it uses Node’s built-in "url" module to parse the URL.
// It takes its pathname, which will be something like /file.txt,
// decodes that to get rid of the %20-style escape codes,
// and prefixes a single dot to produce a path relative
// to the current directory.
function urlToPath(url) {
  var path = require("url").parse(url).pathname;
  return "." + decodeURIComponent(path);
};

// A method to manage GET request. It returns a list of files
// when reading a directory, and the file’s content when reading
// a regular file.
// One tricky question is what kind of Content-Type header we
// should add? Since these files could be anything, our
// server can’t simply return the same type for all of them.
// NPM can  help with that:  the "mime" package knows the correct
// type for a huge amount of file extensions.
methods.GET = function(path, respond){
          // Because it has to touch the disk, and thus might take a while, fs.stat
          // is asynchronous. When the file does not exist, fs.stat will pass
          // an error code property of "ENOENT" to its callback.
        fs.stat(path, function(error, stats){
                // Codes starting with 4 (such as 404) refers to bad requests
                if( error && error.code=="ENOENT") respond(404, "File not found");
                // Codes starting with 5 (such as 500) refers to a server problem
              else if (error) respond(500, error.toString());
            else if (stats.isDirectory())
                      fs.readdir(path, function(error, files){
                        if (error) respond(500, error.toString())
                        else respond(200, files.join("\n"));
                      });
            else
                      respond(200, fs.createReadStream(path), mime.getType(path));
        });
};

methods.POST = function(path, respond, request){
          fs.stat(path, function(error, stats) {
            if( error && error.code=="ENOENT") {
                      const outStream = fs.createWriteStream(path);
                      outStream.on("error", function(error) { respond(500,
 ↪  error.toString()); });
                      outStream.on("finish", function() { respond(204); });
                      // We use pipe to move data from a readable stream to a writable
                       ↪  one,
                      // in this case from the request to the file
                      request.pipe(outStream);
            }
                // Codes starting with 5 (such as 500) refers to a server problem
        else if (error)        respond(500, error.toString());
            else  respond(400, "The file already exists");
          });
};

// The code to handle a delete requests
methods.DELETE = function(path, respond) {
          fs.stat(path, function(error, stats) {
                // You may be wondering why trying to delete a nonexistent file
              // returns a 204 ("no content") status, rather than an error.
              // When the file that is being deleted is not there, you could say
              // that the request's objective is already fulfilled.
              // The HTTP standard encourages people to make requests idempotent.
                if (error && error.code == "ENOENT") respond(204);
            else if (error) respond(500, error.toString());
            else if (stats.isDirectory()) fs.rmdir(path,
             ↪  respondErrorOrNothing(respond));
            else fs.unlink(path, respondErrorOrNothing(respond));
          });
};

function respondErrorOrNothing(respond) {
        return function(error) {
                if (error) respond(500, error.toString());
              else respond(204);
          };
}

// The code for the PUT request
methods.PUT = function(path, respond, request) {
        const outStream = fs.createWriteStream(path);
          outStream.on("error", function(error) { respond(500,
 ↪  error.toString());        });
          outStream.on("finish", function() { respond(204); });
          // We use pipe to move data from a readable stream to a writable one,
          // in this case from the request to the file
          request.pipe(outStream);
};
```

The server acts as a bridge between HTTP and the local file system. It listens for HTTP requests on port 8000 and interprets the **URL path** of each request as a path relative to the server's working directory. This ensures that the server doesn't expose the entire file system, just a local subset of it. Incoming HTTP requests are routed to corresponding handlers for GET, POST, DELETE, and PUT. These handlers interact with the file system asynchronously and use streams wherever possible for efficient data handling. When dealing with files as HTTP resources, different HTTP methods correspond naturally to file operations:

- **GET** is used to **read** a file or list the contents of a directory.
- **POST** and **PUT** are used to **create** or **replace** files.
- **DELETE** is used to **remove** files or directories.

This mapping is a common pattern and it provides a simple interface for interacting with files remotely, for example, to upload data, distribute assets, or manage shared documents.

One of the most important design choices in this server is the use of **streams** when reading from or writing to the file system. For example:

- When a file is read via a GET request, the server uses fs.createReadStream() to **stream the file's content** directly into the response object.
- When writing data with POST or PUT, the incoming request object (which is a readable stream) is **piped** into fs.createWriteStream().

This means that large files are **never fully buffered into memory** at any point. Instead, they are transferred chunk by chunk between disk and network, keeping memory usage low and the system responsive.

We can test this server using Postman:

- **GET** a file: `http://localhost:8000/notes.txt`
- **POST** to create a new file: use `POST http://localhost:8000/newfile.txt` with body content
- **PUT** to overwrite a file
- **DELETE** to remove a file or folder

This project is a great example of how **Node.js's stream-based architecture** allows us to build scalable, efficient servers with minimal resource usage. Would you like to add a section comparing this approach to a synchronous file server or show usage logs from Postman?

### 1.4.2 A Chat Server

So far, we've looked at how Node.js handles HTTP, file systems, and streams. But Node.js is not limited to HTTP: it also allows for **raw TCP communication** using the net module. This capability is especially useful for building custom protocols or lightweight communication systems, such as a **command-line chat server** (see 05.ChatServer.js code):

```
// Include the net library, that contains all the TPC stuff
const net = require('net');

// Create a TCP server
const chatServer = net.createServer();

// A list of client connections
const clientList = [];

// Add a event listener using the on method whenever a "connection event happens, the
↪  listener
// will be called. The event provides a reference to the TPC socket for the new client
chatServer.on('connection', function(client) {
        // decorate the client with an additional property, the name,
    // using the existing properties "remoteAddress" and "remotePort"
    client.name = client.remoteAddress + ':' + client.remotePort;

    // with "write" we can send messages to the new client
    client.write('Hi ' + client.name + '!\n');

    // add the new client to the list
    clientList.push(client);

    // another event listener, it's in the scope of the connection callback method, we
↪   need to access the client.
    // The event is called "data", it is fired each time client sends some data to the
↪   server.
    client.on('data', function(data) { broadcast(data, client) });
});

// A method to send messages to all client, filtering the sender
function broadcast(message, client) {
        for(let i=0; i<clientList.length; i+=1) {
            if(client !== clientList[i]) { clientList[i].write(client.name + " says "
↪   + message); }
          }
}

// The port on which Node is listen to
chatServer.listen(80)
```

The code implements a simple chat server using TCP sockets. WE can test it using a terminal program like telnet or nc (netcat):

```
telnet localhost 9000

nc localhost 9000
```

The server listens for incoming TCP connections. When a client connects, the server assigns them a name (based on their IP and port), sends a greeting, and then starts listening for incoming data using the data event. Whenever a message arrives, the server broadcasts it to all other connected clients. This design demonstrates two key event listeners:

1. A listener for the connection event: triggered whenever a new client connects.
2. A listener for the data event on each client: triggered whenever a client sends a message.

Messages are sent using the .write() method, since TCP sockets are **duplex streams**, meaning they can both read and write. The broadcast function iterates over all clients and sends each message to everyone except the original sender.

However, this initial implementation has a **fatal flaw**: if a client disconnects, the server still tries to write to their socket. This causes errors and can crash the server. We can implement an improved version (see 06.RobustChatServer.js code):

```
const net = require('net');
const chatServer = net.createServer();
const clientList = [];

chatServer.on('connection', function(client) {
  client.name = client.remoteAddress + ':' + client.remotePort
  client.write('Hi ' + client.name + '!\n');
  console.log(client.name + ' joined');
  clientList.push(client);
  client.on('data', function(data) { broadcast(data, client) });

  // Add a new callback for the "end" event: it is fired
  // when a client disconnect. Now when the next client uses the
  // broadcast call, the disconncted client no longer be in the list
  client.on('end', function() {
    console.log(client.name + ' quit');
    clientList.splice(clientList.indexOf(client), 1);
  });

  // adding a callback method for the "error" event, in order to be
  // sure that any erro that occur to clients are logged and the server is no
  // aborted with an exception
  client.on('error', function(e) { console.log(e); });
});

function broadcast(message, client) {
    const cleanup = [];
     for(let i=0; i<clientList.length; i+=1) {
      if(client !== clientList[i]) {
           // adding a check for the write status of the socket during
           // the broadcast call, we can make sure that any sockets that
           // are not available to be written don't cause exception
           if(clientList[i].writable) { clientList[i].write(client.name +
↪ " says " + message); }
           else {
                // make sure that any non writtable socket
            // will be closed and removed by the list.
            // note that we are not removing the client
            // from the list while we are looping throug it
            // in order to avoid side effects
            cleanup.push(clientList[i]);
            clientList[i].destroy();
           }
```

```
                }
        }
        //Remove dead Nodes out of write loop to avoid trashing loop index
          for(let i=0; i<cleanup.length; i+=1)        {
                clientList.splice(clientList.indexOf(cleanup[i]), 1)
          }
}

chatServer.listen(9000)
```

The improved version adds several important features to make the chat server more robust:

- Listening to the end event to detect when a client disconnects and **removing** them from the list.
- Adding a listener for the error event to **gracefully handle socket failures**.
- Checking .writable before writing to a socket to avoid attempting to send data to a dead connection.
- Using a cleanup list to defer removal of broken sockets **after** the main loop, preventing issues with array mutation during iteration.

This kind of defensive programming is essential when working with network code, where failures and disconnects are common.

## 1.5 Express

As applications grow in complexity, managing raw HTTP servers with Node.js can become verbose and error-prone. To address this, the Node.js ecosystem offers **Express**, a minimal and flexible **web application framework** that simplifies the creation of robust web servers and APIs.

Express builds on top of Node's native HTTP module, providing a **more concise and readable syntax**, powerful **routing capabilities**, and an extensible **middleware architecture**. With Express, developers can quickly set up HTTP endpoints, process different kinds of requests, and manage complex flows using a modular and scalable structure. To use Express, it must first be installed via npm:

```
npm install express --save
```

Once installed, a basic "Hello World" web server looks like this:

```
const express = require('express');
const app = express();

app.get('/', function (req, res) {
  res.send('Hello World');
});

const server = app.listen(8081, function () {
  const host = server.address().address;
  const port = server.address().port;
  console.log("Example app listening at http://%s:%s", host, port);
});
```

In this code:

- The express() function creates an app object that acts as the main server.
- The app.get() method sets up a **route** for GET requests on the root URL (/).
- The callback function takes req (request) and res (response) objects, familiar abstractions used throughout Express for handling HTTP transactions.
- The res.send() method sends a simple text response to the client.
- app.listen() binds the server to a port and starts listening for requests.

### 1.5.1 Request and Response Objects

Express extends Node's basic req and res objects with additional utility features. For example:

- req.query, req.params, and req.body to extract data from URLs, routes, and payloads.
- res.send(), res.json(), and res.status() to easily craft responses.

This uniform interface keeps our code clean and expressive, even when dealing with multiple content types or data formats. Express.js is one of the most widely-used frameworks in the Node.js ecosystem. It abstracts the complexity of raw HTTP handling while remaining close to the metal, giving developers fine control over how web servers behave. With powerful routing, intuitive middleware, and a large ecosystem of compatible packages, Express is the go-to choice for building APIs, web apps, and microservices in Node.js.

### 1.5.2 Routing

Routing in Express is **declarative and method-based**: we use functions like app.get(), app.post(), app.put(), and app.delete() to define the server's behavior for different HTTP methods and URLs. This structure creates a **routing table** that determines how requests are handled based on the method and path. Example:

```
app.get('/users', function(req, res) {
    res.send('GET request to /users');
  });

app.post('/users', function(req, res) {
    res.send('POST request to /users');
  });
```

Each route is matched by method and path, and invokes a callback to respond accordingly.

### 1.5.3 Middleware

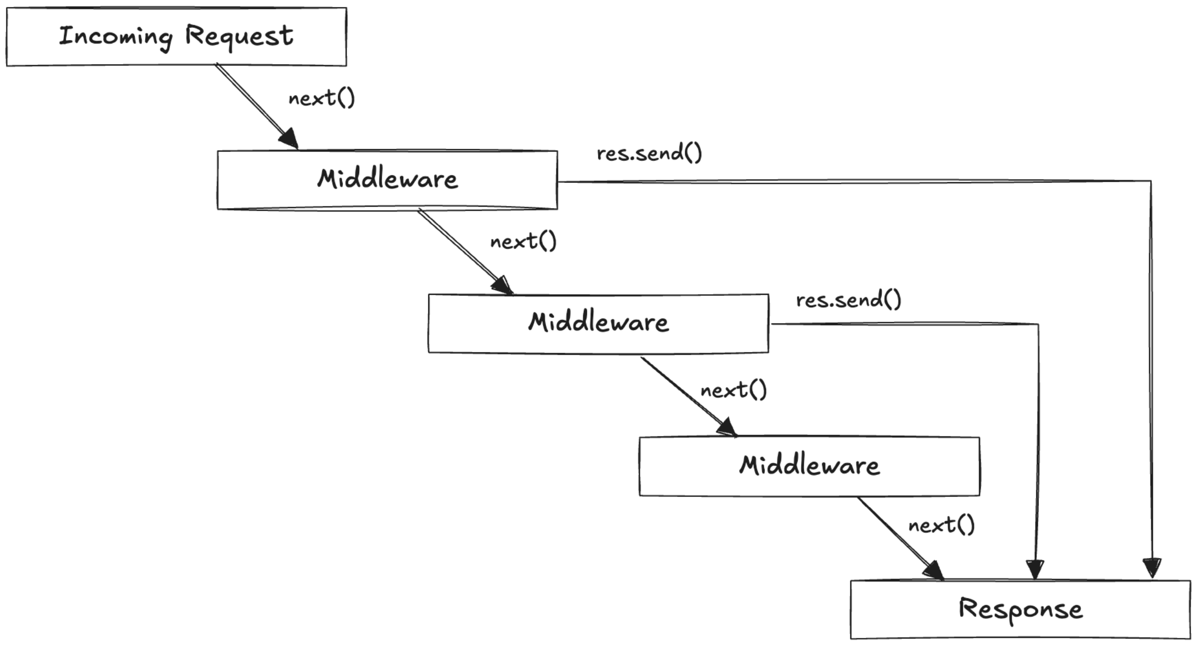
In Express, **middleware** is one of the core concepts that makes the framework flexible, modular, and developer-friendly. Middleware functions are **pieces of code that execute during the lifecycle of an HTTP request**, sitting in between the moment a request arrives and the moment it's handled by a final route like .get() or .post(). A middleware function has access to the request and response objects, and it can either:

- Modify them (e.g., parse a request body),
- Perform actions (e.g., log details, check authentication),
- Or end the request-response cycle by sending a response,
- Or call next() to **pass control to the next middleware** in the chain.

Middleware allows us to **reuse code** for common tasks across multiple routes, keeping our route handlers **clean, focused, and easier to maintain**.They are commonly used for:

- **Logging** every incoming request
- **Authentication and authorization** checks
- **Parsing incoming data**, such as JSON or form submissions
- **Serving static files**
- **Error handling** in a centralized way

Middleware improves both clarity and maintainability by separating responsibilities into modular components:



A request enters the Express app, passes through one or more middleware functions (each optionally transforming or reacting to the request), and eventually reaches the final route handler that sends a response. Each middleware has the opportunity to process or block the request, which gives developers precise control over how HTTP requests are handled.

Middleware is mounted using app.use() or directly within route definitions. For example:

```
app.use(express.json()); // parses JSON payloads automatically
```

As an example, we can use the body-parser module to automatically convert the raw request body into a usable JavaScript object:

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

app.post('/send', bodyParser.json(), function(req, res) {
  // req.body is now available thanks to bodyParser
});
```

In this example:

- When a POST request is made to /send, the bodyParser.json() middleware intercepts the request.
- It reads the raw body, parses the JSON content, and attaches it to req.body.

- Only then does the actual route handler function execute, with full access to the parsed data.

This middleware approach avoids duplicating logic for every route and allows the body-parsing behavior to be reused across the entire application, or selectively added to specific routes.

### 1.5.4 A Simple Twitter Clone

To demonstrate how Express can be used to build real-world web applications, we'll now create a minimal Twitter-like service: a simple app that lets users **post tweets** and **view all submitted tweets**. This example showcases the **core features of Express** — routing, middleware, request parsing, and response formatting — all in under 30 lines of code (see 07.Twitter.js code):

```
// Loading Express, in order to managing HTTP behind the scenes.
const express = require('express');
const bodyParser = require('body-parser');

// Create a server
const app = express();

// Create the array of received tweets
const tweets = [];

// Instead of attaching listener, we can provide methods
// matching HTTP verbs on specific URL requests
app.get('/', function(req, res) { res.send('Welcome to Node Twitter'); })

// A POST route for posting tweets, with a middleware (bodyParser()) to stream
// the post data from the client and turn it in a JavaScript object
// It works only with POST encoded in json
app.post('/send', bodyParser.json(), function(req, res) {
    // express.bodyParser() adds a property to req called body, it
    // contains an object representing the post data
    if (req.body && req.body.tweet) {
                  tweets.push(req.body.tweet);
      res.send( { status:"ok", message:"Tweet received" } );
    }
    else { res.send( {status:"nok", message:"No tweet received" } ); }
})

// a GET route to access all the tweets
app.get('/tweets', function(req,res) { res.send(tweets) });
app.listen(8000);
```

This is the **core functionality**: a route that handles POST requests on /send. It uses bodyParser.json() to automatically parse the request body. If the request contains a tweet field, it adds it to the tweets array and sends a JSON response confirming receipt; and a GET route returning the current array of tweets. Since the data is stored in memory, every new request will reflect the latest state.

To use it:

1. Start the server by running the script.
2. Open a browser and navigate to http://localhost:8000/tweets to view the list of tweets.
3. Use Postman to send new tweets with a POST request to http://localhost:8000/send and a body like:

```
{
    "tweet": "ciao come stai?"
}
```

## 1.6 Why Node.js and IoT?

As we move deeper into the era of the **Internet of Things (IoT)** (where everyday objects become smart and connected) the way we design and build software systems must evolve. IoT applications are characterized by a massive number of devices (sensors, actuators, wearables, gateways) that generate continuous data streams and require low-latency, scalable control systems. In this context, **Node emerges as a highly effective platform** for building IoT backends and services.

### 1.6.1 Scalability and Performance

Node.js is powered by the **Google V8 engine**, which compiles JavaScript into fast machine code. Combined with its **non-blocking, event-driven architecture**, Node.js can handle thousands of concurrent connections efficiently. This is especially important in IoT environments where devices frequently send small payloads of data, often simultaneously, and the backend must remain responsive and scalable.

In contrast to traditional server architectures that spawn new threads per request (consuming memory and CPU), Node.js uses a **single-threaded event loop** and asynchronous callbacks, making it lightweight and ideal for **data-intensive, real-time applications**.

#### 1.6.2 Effective Communication and API Design

IoT systems rely heavily on **APIs** for communication between devices, servers, and dashboards. Node excels at building APIs quickly and efficiently thanks to its rich ecosystem of frameworks like **Express**, **Hapi**, and **Restify**. This model is well-suited for IoT, where devices need to send and retrieve data using lightweight protocols over HTTP. A real-world endorsement comes from companies like **LinkedIn**, which rebuilt its mobile backend using Node.js to take advantage of its **speed and concurrency model**.

#### 1.6.3 Streaming and Real-Time Data Processing

Many IoT devices produce continuous data streams, from temperature sensors to motion detectors or GPS modules. Node handles **streaming data natively**, using its built-in stream interface. This makes it possible to read, process, transform, and forward data from devices **without buffering or storing it in memory**, enabling fast and efficient **data pipelines**. Node also integrates easily with WebSockets or brokers for **real-time bidirectional communication**, which is crucial for use cases such as remote control, live monitoring, and actuation.

#### 1.6.4 Resource Efficiency

IoT environments often involve **constrained devices and edge computing**, small gateways or controllers with limited CPU, memory, or power. Node has a **very low resource footprint** and can run on minimal setups, including single-board computers like the **Raspberry Pi**. This makes it ideal not only for cloud-based processing but also for on-device scripting and edge analytics.

### 1.7 Hands-on Activity

- Improve the file server to create folders. Add support for a method PUT, which should create a directory by calling fs.mkdir
- Improve the Twitter application to provide a support for hashtags. The term in the URL path of a GET is the hashtag, the server must provide just the posts containing that word