Introduction to Node.js

What is Node.js?

Node.js is an open-source, cross-platform JavaScript runtime environment that allows JavaScript code to run outside the browser — specifically on the server-side. It is built on Google Chrome's high-performance **V8 JavaScript engine**.

Traditionally, JavaScript was used only for frontend development, but with Node.js, it became possible to use JavaScript for backend development too, enabling full-stack development using a single language.

Why is Node.js Useful?

- **Fast Execution:** Powered by the V8 engine, JavaScript code runs extremely fast in Node.js.
- **Single Language:** Developers can use JavaScript on both the client and server sides, making development easier and more unified.
- **Non-blocking I/O:** Node.js handles concurrent requests without blocking the main thread, improving scalability.
- **Huge Ecosystem:** The Node Package Manager (NPM) offers thousands of open-source libraries and modules.
- **Real-time Capabilities:** Node is ideal for applications that need real-time interactions like chat apps and games.

Where is Node.js Best Suitable?

Node.js is best suited for applications that are I/O-heavy and require high concurrency but relatively less CPU computation.

- Real-time chat applications (e.g., WhatsApp, Slack)
- RESTful or GraphQL APIs
- Streaming media platforms (e.g., Netflix-like systems)
- Collaborative tools (e.g., Google Docs-style apps)
- IoT systems with many small networked devices
- SPAs and backend services for frontend-heavy apps

Core Concepts in Node.js

1. Event-Driven Architecture

Node.js uses an **event-driven model** where the flow of the program is determined by events — such as user actions, I/O completion, etc. When an event occurs, Node calls the corresponding callback function.

Example: When a file is being read, Node initiates the file read, and once the operation finishes, a callback function is executed with the result.

2. Synchronous vs Asynchronous

- **Synchronous (Blocking):** Tasks are executed one after another. The next task waits until the current one finishes.
- **Asynchronous (Non-Blocking):** Tasks are initiated and continue to run independently. Results are handled later through callbacks or promises.

Node.js favors asynchronous operations, allowing it to handle many requests simultaneously without waiting for each operation to complete before starting the next.

3. Event Loop

The **event loop** is the mechanism that allows Node.js to perform non-blocking I/O operations despite being single-threaded. It continuously checks for events in the queue and executes the corresponding callbacks when the time is right.

It consists of several phases including:

- **Timers:** Executes callbacks scheduled by `setTimeout()` or `setInterval()`.

- **Poll:** Retrieves new I/O events and executes callbacks.
- **Check:** Executes `setImmediate()` callbacks.
- **Close Callbacks:** Executes close events like `socket.on('close')`.

Node.js vs Apache (Traditional Servers)

Feature	Node.js	Apache (with PHP/Java)
Thread Model	Single-threaded with non-blocking I/O	Multi-threaded, blocking I/O
Concurrency	Handles thousands of connections efficiently	Each connection needs its own thread (more memory usage)
Language	JavaScript (full stack)	PHP, Java, or other server-side languages
Performance (I/O bound)	Excellent	Average
Real-Time Apps	Ideal	Needs additional tools or plugins

Suitability for I/O-Intensive Applications

Node.js is extremely efficient for applications that need to perform a lot of I/O operations, such as reading files, accessing databases, handling HTTP requests, or communicating over networks.

Thanks to the event loop and non-blocking nature, Node.js can initiate I/O operations and continue executing other code while waiting for results.

Limitations of Node.js

1. Not Suitable for CPU-Intensive Tasks

Since Node.js is single-threaded, running heavy computations (like video encoding, machine learning, large data parsing) can block the event loop and make the application unresponsive to other users.

Use cases to avoid in Node.js:

- Image or video processing
- Scientific or numerical simulations
- Real-time analytics on massive datasets

2. Callback Hell (now mostly avoidable)

Earlier versions of Node.js relied heavily on nested callbacks, leading to messy code. Modern syntax with Promises and `async/await` has largely solved this.

3. Single Threaded

One long-running operation can block the entire application. Node.js now provides `Worker Threads` to offload CPU-bound tasks, but they are not as straightforward as multi-threading in Java or C++.

Conclusion

Node.js is a powerful tool for building fast, scalable, and real-time applications, particularly when dealing with I/O-bound workloads. However, it should be used carefully for CPU-bound or computation-heavy applications where other platforms may perform better.

Building Node web applications

servers using JavaScript. However, the approach is relatively low-level, giving more control but requiring deeper understanding.

Node.js was designed specifically for building scalable server-side and networking applications.

Core Networking Modules in Node.js

Node.js provides several core networking modules to facilitate web application development:

- `net (require('net'))`: Provides TCP socket support for creating servers and clients.
- `dgram (require('dgram'))`: Supports the creation of UDP/Datagram sockets for connectionless data transfer.
- `http (require('http'))`: Supports HTTP server and client functionality for web requests.
- `https (require('https'))`: Similar to http but with SSL/TLS encryption for secure communication.

To get started, we will use the http module to build a simple web server that will respond with "hello world" to the client.

HTTP Server

Node provides a relatively low-level API, and its HTTP interface reflects that. Compared to higher-level frameworks or languages like PHP, Node's approach is more minimal to maintain speed and flexibility.

Node provides HTTP server and client interfaces through the http module

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

The `http` module in Node.js offers the `createServer` function, which is used to create an HTTP server. It accepts a single callback function that's invoked each time the server receives an HTTP request. This callback receives two arguments—the request and response objects—commonly referred to as `req` and `res`.

```
var http = require('http');
const server = http.createServer(function(request, response){
    // handle request
});
```

`Request Object` **lets the server read and handle what the client is asking for.**

`Response object` **used to send data back to the client in response to their HTTP request.**

Node doesn't automatically send a response to the client.

```
// Send response
response.write('Hello, World!');
```

Once the request callback runs, it's up to you to end the response using `response.end()`.

```
// End response
response.end();
```

This gives you the flexibility to perform async operations before responding. If you forget to call `response.end()`, the request will hang or stay open until the client times out. Since Node servers handle many requests over time, properly ending each response is crucial.

To start the HTTP server, call its `listen` method and specify the desired port number.

A basic HTTP server that responds with Hello World!

Here's an example of a simple server using the http module:

```
const http = require('http'); // import http module

const server = http.createServer(function(req, res){
});

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

This server listens on port 3000 and responds with "**Hello, World!**" to every HTTP request.

Testing the Server

To verify that your HTTP server is working correctly, follow these steps:

1. **Start the Server**
   Run the following command in your terminal to start the server:
   `node server.js`
   (Assuming your code is saved in a file named server.js)
   You should see this message in the terminal:
   `Server is running at http://localhost:3000/`
2. **Test the Connection**
   Using a Web Browser:
   Open your browser and go to http://localhost:3000/. You should see "Hello, World!" displayed.

For each HTTP request the server receives, the request callback function is executed with fresh `request` and `response` objects.

1. **The Request Object:** The request object holds all the information about the client's HTTP request. This is an instance of `IncomingMessage`.

   When an HTTP request is received, Node.js automatically parses the request up to the HTTP headers. Means node will take the information from the incoming HTTP request such as the method, URL, and headers and attach them as properties to the req object, so you, as the developer, can easily access them when handling the request.

   Request Object includes:

   - **HTTP method** (like GET, POST)
   - **URL** or route the client is trying to access
   - **Headers** sent by the client
   - **Data** sent in the body (like form submissions or JSON body, in POST requests)

   However, Node doesn't automatically parse the request body (e.g., form data or JSON) - you'd need to handle that manually.

   This `Request object` lets the server read and handle what the client is asking for.

2. **The Response Object:**The response object used to send data back to the client in response to their HTTP request. This is an instance of `ServerResponse`. It allows the server to:

   - Set the **status code** (like 200 for success or 404 for not found)
   - Set **headers** (like Content-Type to tell the browser what kind of data is being sent)
   - **Write content** (such as HTML, JSON, or plain text)
   - **End the response** when everything is sent

   This `Response Object` lets the server build and send the appropriate response to the client's request.

Accessing the Request Object (`req`) in Node.js

The `req` object contains all the information about the incoming HTTP request. Here's how you can access its key properties:

1. Request Method

Returns the HTTP method used for the request.
→ Example: `'GET'`, `'POST'`, `'PUT'`, `'DELETE'`

2. Request URL

```
req.url
```

Returns the path or URL requested by the client.
→ Example: `'/home'`, `'/api/users'`

3. Request Headers

```
req.headers
```

Returns an object containing all the request headers.

→ Example:

```
{
  'host': 'localhost:3000',
  'user-agent': 'Mozilla/5.0',
  'content-type': 'application/json'
}
```

You can also access specific headers like this:

```
const contentType = req.headers['content-type'];
```

4. Request Body

Note: The body is *not available by default* in plain Node.js.

To access the request body (e.g., for POST or PUT requests), you need to manually read the data stream:

```
let body = '';
req.on('data', chunk => {
  body += chunk;
});

req.on('end', () => {
  console.log('Request Body:', body);
});
```

Alternatively, in frameworks like **Express**, you can use middleware like `express.json()` or `body-parser` to automatically parse the body.

5. Full Example

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

const server = http.createServer((req, res) => {
  console.log('Method:', req.method);
  console.log('URL:', req.url);
  console.log('Headers:', req.headers);

  let body = '';
  req.on('data', chunk => {
```

```
  req.on('end', () => {
    console.log('Body:', body);

    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Request received');
  });
});

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

This example logs method, URL, headers, and body of the incoming request, then sends a response back to the client.

Testing the Server

1. GET request:

```
curl http://localhost:3000
```

or you can type below url in your browser.

```
http://localhost:3000
```

Using `curl` in Terminal

You can not send a raw **POST request with a body** directly from the browser's address bar. The browser address bar can only issue **GET requests**.

To test **POST requests with a body** (like form data or JSON), we use tools like:

- **curl** (command-line tool)
- **Postman** (GUI tool)
- **JavaScript** (`fetch` or `Axios`) in frontend code

Below are examples for different types of requests:

2. POST request with body (form data):

```
curl -X POST -d "name=John&age=2." http://localhost:3000
```

3. POST request with JSON:

```
curl -X POST -H "Content-Type: application/json" -d '{"name":"John","age":"2"}' http://localhost:3000
```

Understanding the `res` Object in Node.js

The `res` object in Node.js is used to control and send the HTTP response back to the client. You can set headers, status codes, write content, and end the response using various methods.

1. Setting the Status Code

```
res.statusCode = 200;
```

[illegible]

2. Setting Response Headers

```
res.setHeader('Content-Type', 'text/plain');
```

Sets a specific HTTP header on the response.
→ Common headers include `Content-Type`, `Content-Length`, `Cache-Control`, etc.

3. Writing Data to the Response

```
res.write('Hello, world!');
```

Sends a chunk of the response body to the client. You can call this multiple times to send data in parts.

4. Ending the Response

```
res.end();
```

Signals that the response is complete.

Optionally, you can combine `res.write()` and `res.end()` into a single `res.end('data')` call, which acts as a shorthand and simplifies the code.

```
res.end('Hello, world!');
```

Important: You must always call `res.end()` to properly close the response. If not, the request will hang until the client times out.

5. Retrieving and Removing Headers

Get a Header Before Sending

```
const contentType = res.getHeader('Content-Type');
```

Retrieves the value of a response header before the headers are sent.

Remove a Queued Header

```
res.removeHeader('Content-Encoding');
```

Removes a previously set header before it's sent to the client.

6. Sending Headers Explicitly

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

Sends the response status code and headers explicitly. This method transitions the response into "body mode" and ensures all headers are sent before writing the response body.

7. Adding Custom Headers

```
res.setHeader('X-Custom-Header', 'MyCustomValue');
```

You can add custom headers to your HTTP response using `setHeader`. Custom headers usually begin with `X-`, but it's not mandatory.

8. Full Example with Headers

```
http.createServer((req, res) => {
  // Set standard and custom headers
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Powered-By', 'Node.js');
  res.setHeader('X-Custom-Header', 'MyCustomValue');

  // Retrieve a header
  const contentType = res.getHeader('Content-Type');
  console.log('Content-Type:', contentType);

  // Remove an unnecessary header
  res.removeHeader('Content-Encoding');

  // Send headers explicitly
  res.writeHead(200, {
    'Content-Type': 'text/html',
    'X-Custom-Header': 'MyCustomValue'
  });

  // Send response body
  res.end('<h1>Hello with custom headers!</h1>');
}).listen(3000, () => {
  console.log('Server is running at http://localhost:3000');
});
```

This example demonstrates how to set, retrieve, remove, and send headers explicitly before writing the response body.

Serving Static Files in Plain Node.js

What Are Static Files?

Static files are files that don't change dynamically on the server. They are delivered to the browser exactly as they are stored on the server. These include:

- HTML files
- CSS stylesheets
- JavaScript scripts
- Images (JPG, PNG, SVG)
- Fonts and videos

When a user visits a website, these files are sent from the server to the user's browser, where they are rendered or executed.

Node.js doesn’t come with built-in static file handling like some web frameworks (e.g., Express). So if you want to serve static files in plain Node.js, you need to:

1. Read the request URL
2. Locate the requested file on the server
3. Read the file from the filesystem
4. Send its contents back in the response

Creating a Static File Server

1. Define a Root Directory

Each file server has root directory. In server define `root` variable, which will act as the static file server's root directory.

It's very useful when working with files because it ensures you always reference paths correctly, no matter where your script is executed from.

If your static files are stored inside a public folder within the same directory as your Node server.js file, you should use the path.join() method to build the absolute path properly.

Here's how you'd define the root variable:

```
const path = require('path');
const root = path.join(__dirname, 'public');
```

2. Parse the URL Path

Use Node's url module to extract the path from the request.

```
const url = require('url');
const parsedUrl = url.parse(req.url);
let pathname = parsedUrl.pathname;
```

If the user visits /, serve a default file like index.html:

```
if (pathname === '/') {
  pathname = '/index.html';
}
```

Then join it with the root directory to create the full path:

```
const filePath = path.join(root, pathname);
```

3. Read the File with a Stream

Now we'll use Node's `fs.createReadStream()` to open the file and read its contents in chunks.

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

let body = '';
const stream = fs.createReadStream(filePath);

stream.on('data', chunk => {
  body += chunk;
});

stream.on('end', () => {
  console.log('Request Body:', body);
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end(body);
});
```

Here, the data event collects each chunk of the file, and the end event signals that the whole file is ready. Once complete, the content is written to the response.

Full Example: Basic Static File Server

```
const http = require('http');
const fs = require('fs');
const path = require('path');
const root = path.join(__dirname, 'public');

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url);
  let pathname = parsedUrl.pathname;

  if (pathname === '/') {
    pathname = '/index.html';
  }

  const filePath = path.join(root, pathname);

  let body = '';
  const stream = fs.createReadStream(filePath);

  stream.on('data', chunk => {
    body += chunk;
  });

  stream.on('end', () => {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(body);
  });
});

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

Summary

Step	What Happens
1️⃣	The root directory (`public/`) is defined as the base for all file serving
2️⃣	The request URL is parsed to get the file path
3️⃣	That path is joined with the root to find the full location on disk
4️⃣	The file is read using `fs.createReadStream()`
5️⃣	The file contents are collected and sent back to the browser with `res.end()`

1. Folder Structure

Make sure your project folder looks like this:

```
staticFileServer/
├── server.js          ← your Node.js server file
└── public/
  ├── index.html      ← home page
  └── about.html      ← about page
```

2. HTML File Contents

`public/index.html`

`public/about.html`

```
<h1>This is About Page</h1>
```

Testing Procedure

Step 1: Run the Server

In your terminal, run the server using the following command:

```
node server.js
```

You should see the output:

```
Server running at http://localhost:3000
```

Step 2: Open Browser and Visit the Following URLs

✅ Test 1: Home Page

- **Open:** `http://localhost:3000/`
- **Output:** You should see

  ```
  This is Home Page
  ```

✅ Test 2: About Page

- **Open:** `http://localhost:3000/about.html`
- **Output:** You should see

  ```
  This is About Page
  ```

❌ Test 3: Invalid Page (e.g., login.html)

- **Open:** `http://localhost:3000/login.html`
- **Output:** Browser shows: **Unable to Connect**
- Here we are not handling error. If handled properly that error will be displayed like `404 Not Found`

Optimized Data Transfer Using stream.pipe()

In Node.js, when serving files or handling large data, the most efficient way to transfer data is by using the .pipe() method from streams. This method connects a readable stream (like a file) directly to a writable stream (like the HTTP response).

```
const http = require('http');
const fs = require('fs');
const path = require('path');
const url = require('url');

const root = path.join(__dirname, 'public');

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url);
  let pathname = parsedUrl.pathname;

  if (pathname === '/') {
    pathname = '/index.html';
```

```
  const filePath = path.join(root, pathname);

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

  const stream = fs.createReadStream(filePath);
  stream.pipe(res);
});

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

Static File Server with Error Handling

Adding error handling in a Node.js static file server is important for creating a reliable and user-friendly application. Without it, your server may crash or behave unpredictably when users request files that don’t exist or can’t be accessed. With proper error handling, the server can catch these issues and respond with appropriate HTTP messages, such as a 404 Not Found, instead of leaving users with blank pages or browser errors.

```
const http = require('http');
const fs = require('fs');
const path = require('path');
const url = require('url');

const root = path.join(__dirname, 'public');

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url);
  let pathname = parsedUrl.pathname;

  // Default to index.html if root is requested
  if (pathname === '/') {
    pathname = '/index.html';
  }

  const filePath = path.join(root, pathname);

  // Check if file exists and handle errors
  const stream = fs.createReadStream(filePath);

  stream.on('error', () => {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('404 Not Found');
  });

  // Set the content type (defaulting to HTML here for simplicity)
  res.writeHead(200, { 'Content-Type': 'text/html' });
  stream.pipe(res);
});

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

In this program, if the requested file does not exist, the `fs.createReadStream()` throws an error, but since `res.writeHead(200)` is called before the error is handled, the server may crash or behave unpredictably. This can cause the browser to show a generic "**Unable to connect**" error (especially in Firefox) instead of a proper "**404 Not Found**" response.

We need to make sure you only send headers after the stream is successfully opened. That's because when the file doesn't exist, the stream will emit an 'error' event, and if you already sent a 200 OK response, it's too late to change it.

```
const http = require('http');
const fs = require('fs');
const path = require('path');
const url = require('url');

const root = path.join(__dirname, 'public');

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url);
  let pathname = parsedUrl.pathname;

  // Default to index.html if root is requested
  if (pathname === '/') {
    pathname = '/index.html';
  }

  const filePath = path.join(root, pathname);

  // Check if file exists and handle errors
  const stream = fs.createReadStream(filePath);

  stream.on('error', () => {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('404 Not Found');
  });

  //only send headers after the stream is successfully opened
  stream.on('open', () => {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    stream.pipe(res);
  });
});

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

Preemptive Error Handling with fs.stat in Node.js

In a Node.js static file server, instead of waiting for a file read error to occur when using `fs.createReadStream`, you can proactively check if the file exists and is accessible before trying to read it. This approach is known as preemptive error handling, and it can be done using the `fs.stat()` or `fs.promises.stat()` method.

What is `fs.stat()`?

The `fs.stat()` function checks the status of a file or directory. It tells you:

- Whether the file exists
- If it's a file or directory
- File size, modified time, permissions, etc.

Why Use Preemptive Error Handling?

- Avoids stream errors before they happen

- Improves response time, as you're not starting a stream that will fail
- Allows you to check file types before serving (e.g., skip directories)

```
const http = require('http');
const fs = require('fs');
const path = require('path');
const url = require('url');

const root = path.join(__dirname, 'public');

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url);
  let pathname = parsedUrl.pathname === '/' ? '/index.html' : parsedUrl.pathname;
  const filePath = path.join(root, pathname);

  // Check file existence and type before streaming
  fs.stat(filePath, (err, stats) => {
    if (err) {
      if (err.code === 'ENOENT') {
        // File not found
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('404 Not Found');
      } else {
        // Some other file system error
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end('500 Internal Server Error');
      }
      return;
    }

    // Serve the file
    res.writeHead(200, { 'Content-Type': 'text/html' });
    fs.createReadStream(filePath).pipe(res).on('error', () => {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end('500 Internal Server Error');
    });
  });
});

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

We've covered a solid foundation of Node.js by working with low-level HTTP servers and serving static files. But as applications grow, handling everything manually in plain Node can get complex and repetitive.

Here are some important topics that typically come next when building full-featured web apps:

- Routing
- Handling form data (GET and POST requests)
- Connecting to databases (e.g., MongoDB, MySQL)
- Using middleware
- Working with templating engines (like EJS)
- Handling file uploads
- Managing sessions and authentication
- Building RESTful APIs

- Error handling

We will cover all these topics easily and efficiently using Express.js, a powerful and lightweight web framework for Node.js. Let’s get started!