@Emmersive Learning

HANDBOOK

2024

MEHAMMED TESHOME

NodeJS Handbook

by : Mohammed Teshome

Learn with us :

Telegram: https://t.me/EmmersiveLearning

Youtube: https://www.youtube.com/@EmmersiveLearning/

Table of content

Chapter 0 : Roadmap to Learn Node Js

Here's a **comprehensive roadmap for mastering Node.js**, covering everything from basics to advanced topics. This roadmap ensures a solid foundation and prepares you to build real-world applications.

---

1. Basics of Node.js

Objective: Understand what Node.js is and get comfortable with its environment.

- What is Node.js?
- Setting up the environment:
  - Installing Node.js and npm.
  - Running Node.js programs.
- Understanding Node.js architecture: Event loop, single-threaded nature, and non-blocking I/O.
- Working with global objects (`console`, `__dirname`, `__filename`).
- Using the Node.js REPL.

---

2. Core Concepts

Objective: Master Node.js built-in modules and functionality.

- **Modules:**
  - Built-in modules (e.g., `fs`, `http`, `path`, `os`).

    - Creating custom modules.
    - Using `require` and `module.exports`.
- **File System (fs):**
    - Reading and writing files synchronously and asynchronously.
    - Managing directories.
- **Events and EventEmitter:**
    - Event-driven programming.
    - Creating and handling custom events.
- **Buffers and Streams:**
    - Reading and writing data streams.
    - Using `Readable`, `Writable`, and `Duplex` streams.
- **HTTP Module:**
    - Creating basic HTTP servers.
    - Handling HTTP requests and responses.

---

3. Asynchronous Programming

Objective: Learn how Node.js handles async tasks efficiently.

- Callbacks and callback hell.
- Promises:
    - Creating and chaining promises.
    - `Promise.all` and `Promise.race`.
- Async/Await:
    - Writing cleaner asynchronous code.
    - Error handling with `try...catch`.
- Understanding the Event Loop and task scheduling.

---

4. Package Management

Objective: Work with npm and manage dependencies.

- Initializing a Node.js project (`npm init`).
- Installing and removing packages (`npm install`, `npm uninstall`).
- Global vs local packages.
- Semantic versioning.
- Using `npx` to run CLI tools.

---

5. Express.js Framework

Objective: Build robust and scalable web applications.

- Introduction to Express.js.
- Creating a basic Express server.
- Middleware:
  - Built-in, third-party, and custom middleware.
  - `req`, `res`, and `next`.
- Routing:
  - Handling different HTTP methods (GET, POST, PUT, DELETE).
  - Route parameters and query strings.
- Static files and serving resources.

---

6. Building RESTful APIs

Objective: Develop RESTful APIs with proper standards.

- Designing RESTful APIs (CRUD operations).
- Using query parameters, route parameters, and body data.
- JSON responses and status codes.
- Error handling in APIs.
- Versioning APIs.

---

7. Working with Databases

Objective: Integrate Node.js with databases for dynamic data management.

- **NoSQL (MongoDB):**
  - Introduction to MongoDB and Mongoose.
  - Connecting to MongoDB.
  - Creating schemas and models.
  - Performing CRUD operations.
- **SQL (PostgreSQL/MySQL):**
  - Connecting to relational databases using libraries like `pg`.
  - Writing queries.
  - Performing CRUD operations.

---

8. Authentication and Authorization

Objective: Secure your applications.

- User authentication:
  - Hashing passwords using `bcrypt`.

    - JSON Web Tokens (JWT).
    - Session-based authentication.
- Authorization:
    - Role-based access control (RBAC).
    - Middleware for protected routes.

---

9. Advanced Topics

Objective: Gain deeper insights into Node.js internals and advanced techniques.

- **Real-time Applications:**
    - Using `Socket.IO` for WebSocket communication.
    - Building chat apps and live updates.
- **Error Handling:**
    - Using try-catch blocks and centralized error handling.
    - Debugging tools like `console` and `node-inspect`.
- **Performance Optimization:**
    - Understanding clustering and worker threads.
    - Load balancing.
    - Caching with Redis or in-memory caching.
- **File Uploads:**
    - Handling file uploads with `multer`.
- **Environment Variables:**
    - Using `dotenv` for configuration.

---

10. Testing and Debugging

Objective: Write reliable and bug-free code.

- Unit testing with `Mocha` or `Jest`.
- Integration testing for APIs.
- Mocking and stubbing dependencies.
- Debugging Node.js applications.

---

11. Deployment and Production

Objective: Host your application for users.

- Deployment platforms:
    - Deploying on Heroku, AWS, Vercel, or DigitalOcean.
- Process management with `PM2`.
- Using `nginx` for reverse proxy.
- Securing apps with HTTPS and SSL.
- Logging and monitoring:
    - Using tools like `winston` or `morgan`.

---

12. Real-world Projects

Objective: Apply everything you've learned.

- Build a basic RESTful API (e.g., a to-do app).
- Develop a real-time chat application with `Socket.IO`.
- Create a full-stack app with Express and a database.

- Build a scalable e-commerce backend with Node.js.

---

13. Exploring Further

Objective: Stay up-to-date and deepen expertise.

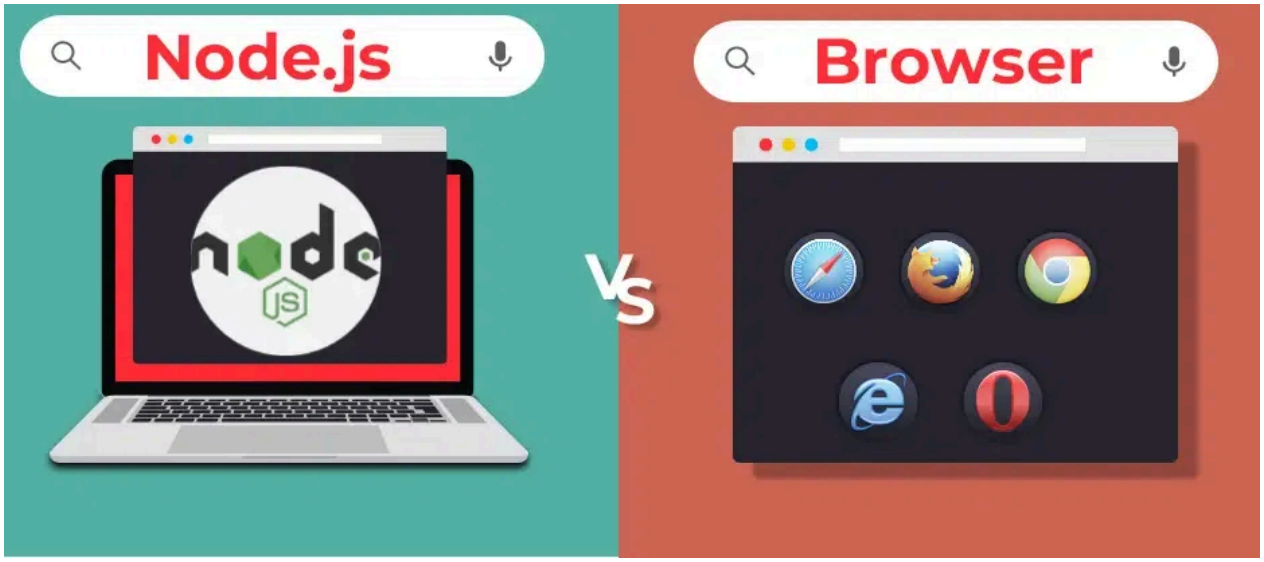
- Learn advanced frameworks like Nest.js.
- Contribute to open-source Node.js projects.
- Follow Node.js best practices for clean code and scalability.

Let's start with **Chapter 1: The Basics of Node.js** in detail.

Chapter 1: Introduction to Node.js

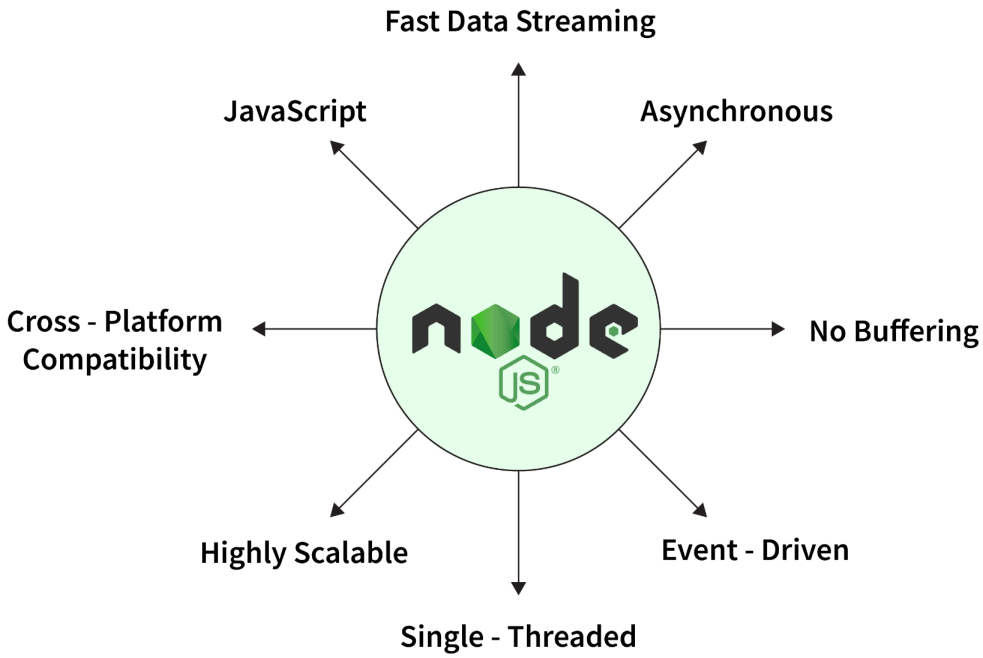
1. What is NodeJS?

Node.js is a **runtime environment** that allows you to run JavaScript outside the browser. It's widely used for server-side development, allowing you to build fast, scalable, and efficient applications.



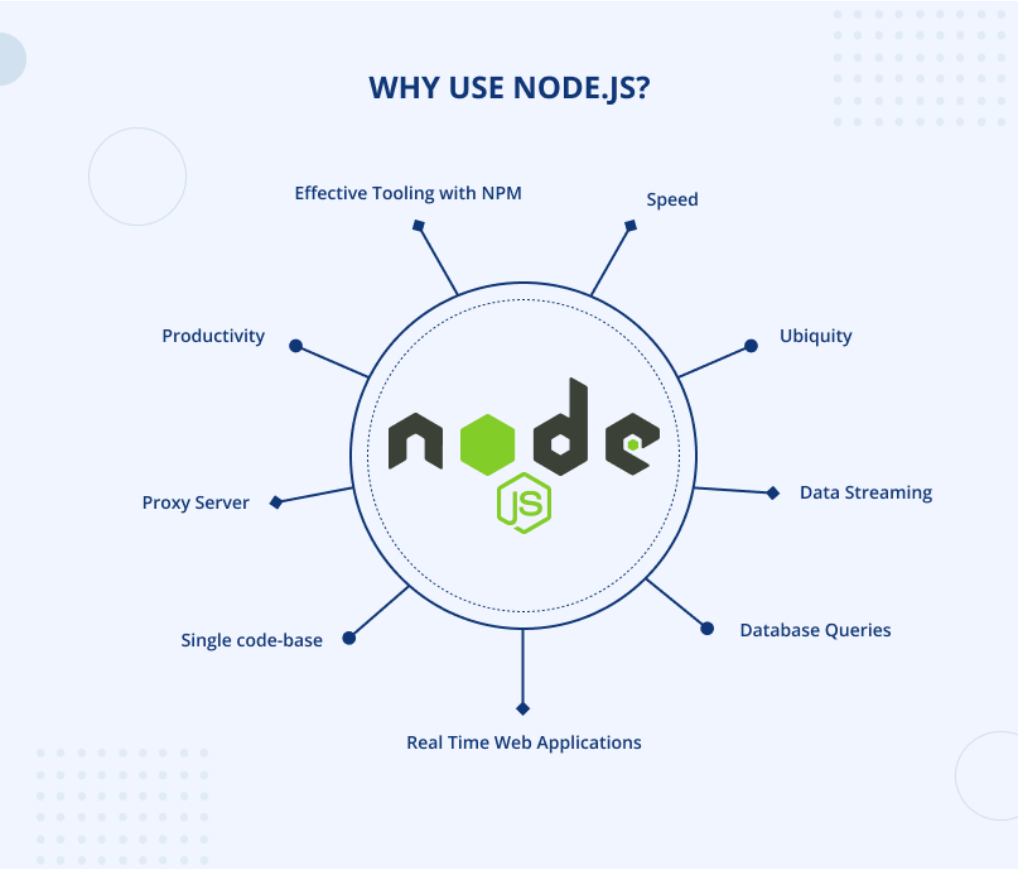
Key Features:

1. **Event-driven:** Handles requests using events and callbacks.
2. **Non-blocking I/O:** Performs tasks asynchronously.
3. **Single-threaded:** Uses one thread to handle multiple concurrent requests.



Why Use Node.js?

- Efficient for I/O-intensive applications like APIs, real-time chats, or data streaming.
- Same language (JavaScript) for both client-side and server-side.



Application of nodejs

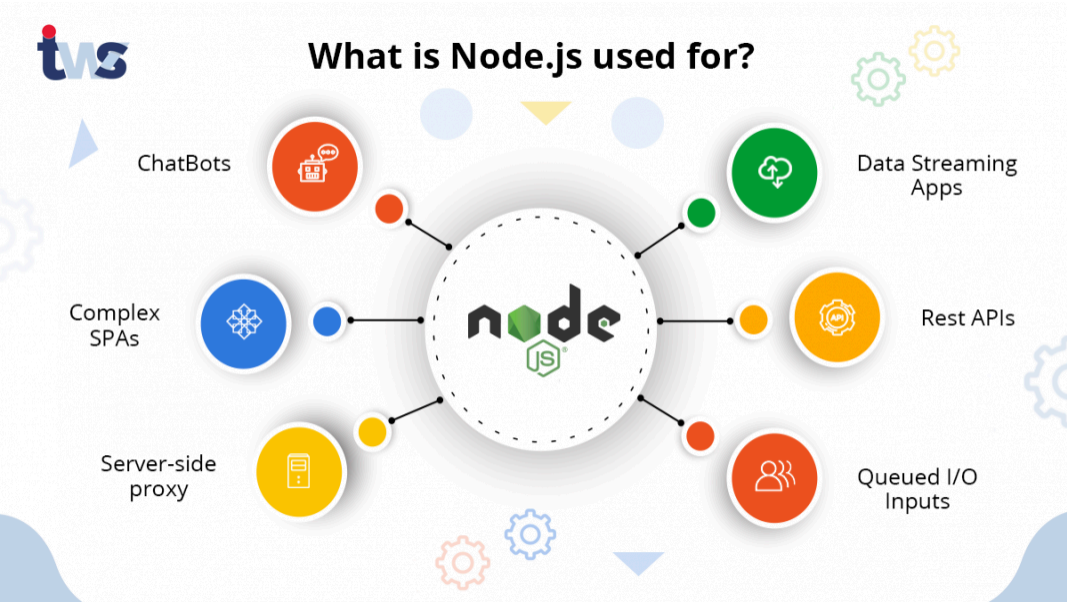
Examples of Nodejs Applications

- Streaming applications
- Chat applications

- Command-line applications
- Browser games
- Embedded systems

The usage of the Nodejs platform is not only limited to web application development but also u
applications including:

- Microservices
- Scripting
- Animations
- APIs Development
- Backends and Frontends
- Servers



---

2. Setting Up Node.js

1. **Install Node.js:**
    - Download it from the Node.js official website.
    - Choose the **LTS (Long-Term Support)** version for stability.
2. **Verify Installation:**

Open the terminal (or command prompt) and check:

node -v

npm -v

- This confirms Node.js and npm (Node Package Manager) are installed.

---

3. Running Your First Node.js Program

```
Create a file named app.js:
 console.log("Hello, Node.js!");
```

Run the file in the terminal:

```
node app.js
```

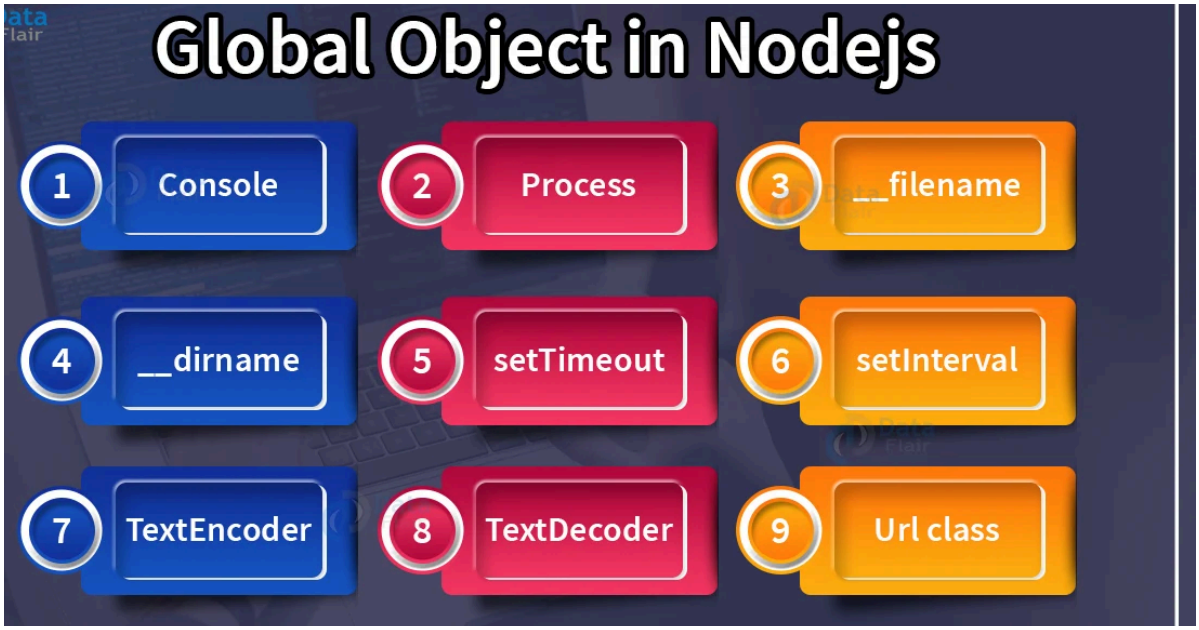
Output:

```
Hello, Node.js!
```

---

4. Core Node.js Concepts

(a) Global Objects



Node.js provides several global objects that can be accessed anywhere.

`console`: Logs messages to the console.

```
console.log("Welcome to Node.js");
```

`__dirname`: Gives the directory name of the current file.

```
console.log(__dirname);
```

`__filename`: Gives the full path of the current file.

```
console.log(__filename);
```

-

(b) Node.js REPL

REPL (Read-Eval-Print-Loop) is an interactive shell for running JavaScript.

Start the REPL:

```
Node
```

Type commands:

```
> console.log("Testing REPL");
```

Exit REPL:

```
.exit
```

---

5. Working with Node.js Modules

Modules allow you to break your application into reusable pieces of code.

(a) Built-in Modules

Example: Using the `os` module:

```
const os = require('os');
console.log("Platform:", os.platform());
console.log("Architecture:", os.arch());
```

(b) Creating Custom Modules

Create a file `greet.js`:

```
module.exports = function (name) {
    return `Hello, ${name}!`;
};
```

Use the module in `app.js`:

```
const greet = require('./greet');
console.log(greet("Alice"));
```

Output:

```
Hello, Alice!
```

---

6. File System (fs) Module

The `fs` module lets you work with the file system.

Write to a file:

```
const fs = require('fs');
fs.writeFileSync('example.txt', 'Hello, Node.js!');
```

Read from a file:

```
const data = fs.readFileSync('example.txt', 'utf8');
console.log(data);
```

---

7. Creating a Simple HTTP Server

The `http` module lets you create web servers.

Example:

```
Create a file named server.js:
 const http = require('http');

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

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

1. Run the server:

```
 node server.js
```

2. Visit `http://localhost:3000` in your browser to see:

```
Hello, World!
```

---

8. Key Takeaways

- Node.js uses JavaScript for server-side programming.
- Its asynchronous, non-blocking nature makes it ideal for high-performance applications.
- Built-in modules like `fs`, `http`, and `os` are powerful tools for development.
- You can write modular, reusable code using custom modules.

Next Steps

- Experiment with built-in modules like `fs`, `path`, and `http`.
- Create small projects (e.g., a file writer or a simple server).
- Let me know if you want to dive deeper into any of these topics before moving to **Chapter 2: Core Concepts**!

Chapter 2: Nodejs Modules

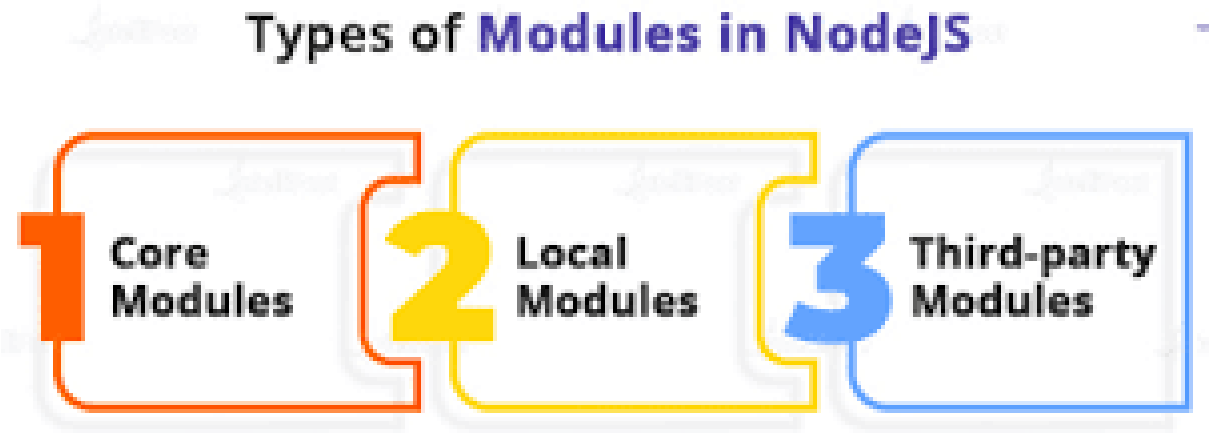
What Are Modules in Node.js?

A **module** in Node.js is a reusable piece of code that encapsulates related functionality. Node.js uses modules to:

- Organize code into smaller, manageable pieces.
- Share functionality between files.
- Enhance reusability and maintainability.

Node.js has **three types of modules**:

1. **Built-in modules**: Provided by Node.js (e.g., `fs`, `http`).
2. **User-defined modules**: Custom modules you create.
3. **Third-party modules**: Installed via npm (e.g., `express`)



How Node.js Modules Work

Node.js follows the **CommonJS module system**.
Key concepts:

- **`require`:** Used to import a module.
- **`module.exports`:** Used to export functionality from a module.

Module Wrapper Function

Every module in Node.js is wrapped in a function like this:

Javascript code

```
(function (exports, require, module, __filename, __dirname) {
    // Module code here
});
```

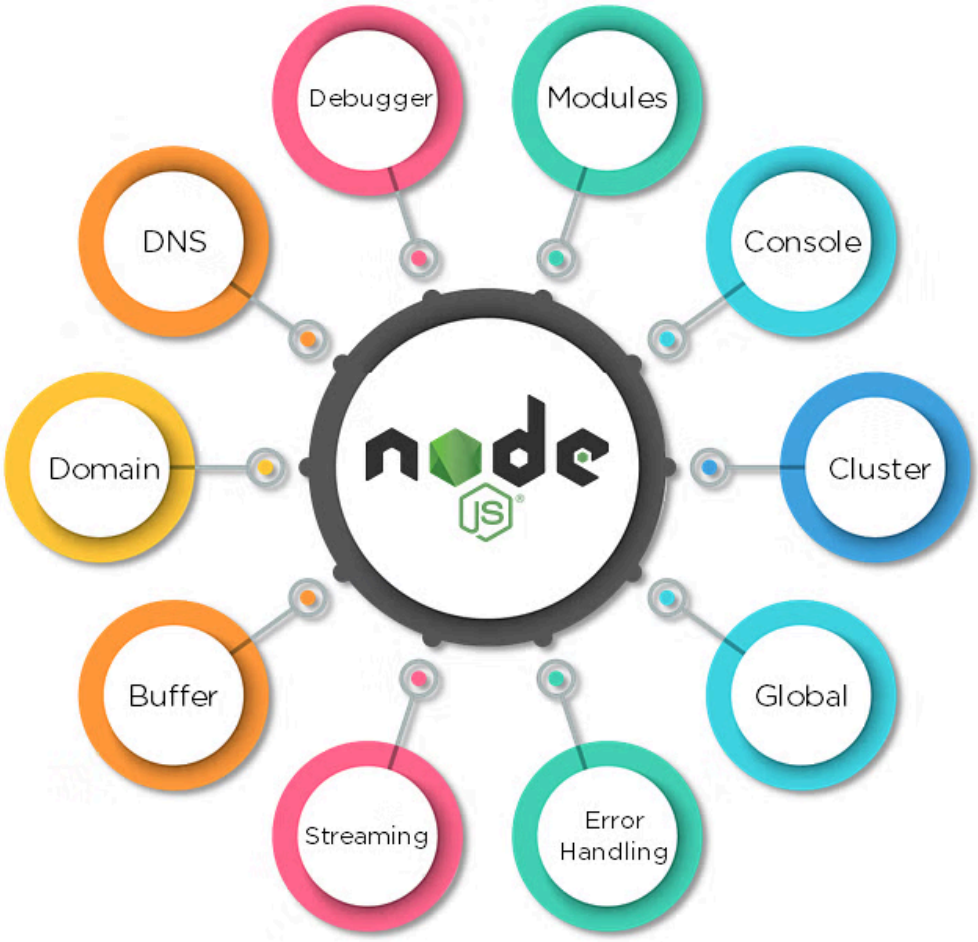
This gives access to:

- **`exports`**: Shortcut for exporting objects.
- **`require`**: Function to import modules.
- **`module`**: Reference to the module object.
- **`__filename`**: Absolute path of the current file.
- **`__dirname`**: Directory path of the current file.

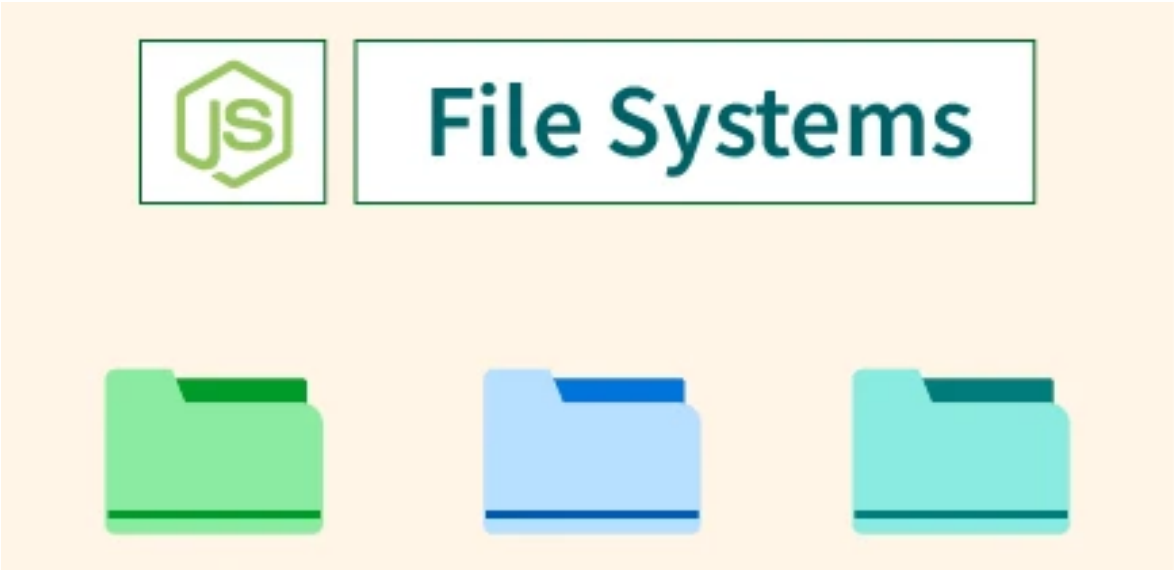
---

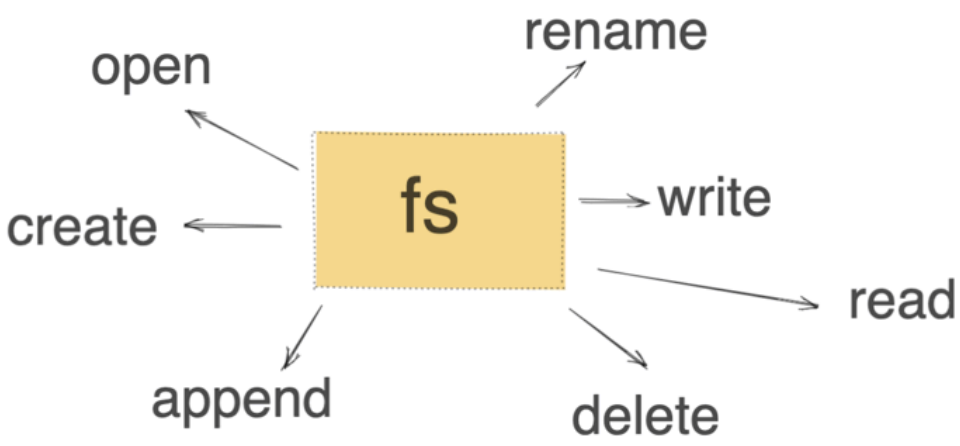
1. Built-in Modules

Node.js provides numerous built-in modules. Some commonly used ones:



(a) File System (`fs`)





Write to a file:

```
const fs = require('fs');
fs.writeFileSync('example.txt', 'Hello, Node.js!');
```

Read from a file:

```
const data = fs.readFileSync('example.txt', 'utf8');
console.log(data);
```

(b) Path (`path`)

Useful for handling and transforming file paths.

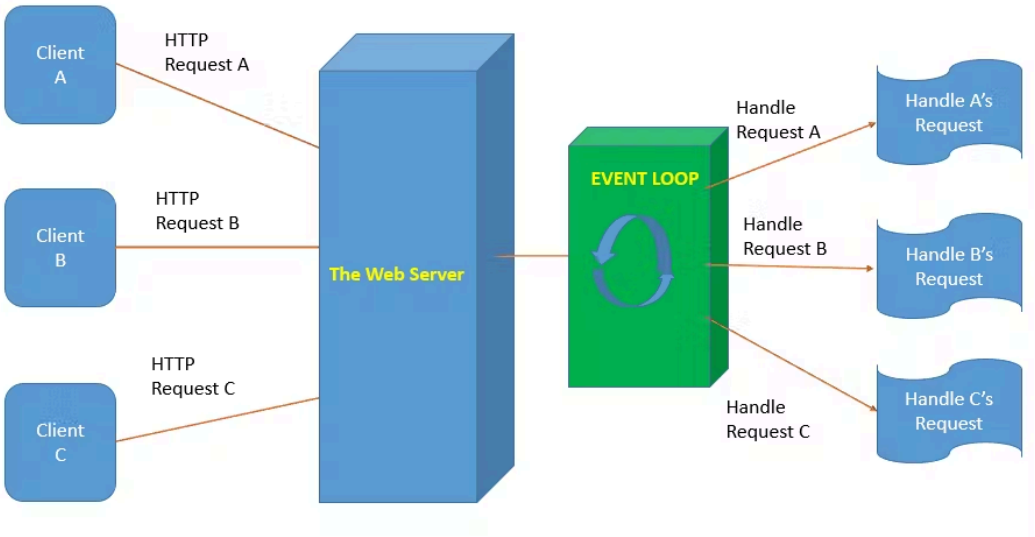
Path Properties and Methods

Method	Description
basename()	Returns the last part of a path
delimiter	Returns the delimiter specified for the platform
dirname()	Returns the directories of a path
extname()	Returns the file extension of a path
format()	Formats a path object into a path string
isAbsolute()	Returns true if a path is an absolute path, otherwise false
join()	Joins the specified paths into one
normalize()	Normalizes the specified path
parse()	Formats a path string into a path object
posix	Returns an object containing POSIX specific properties and methods
relative()	Returns the relative path from one specified path to another specified path
resolve()	Resolves the specified paths into an absolute path
sep	Returns the segment separator specified for the platform
win32	Returns an object containing Windows specific properties and methods

```
const path = require('path');
console.log(path.basename(__filename)); // Current file name
console.log(path.dirname(__filename)); // Directory name
console.log(path.extname(__filename)); // File extension
```

(c) HTTP

For creating servers and handling requests.




```
const http = require('http');
const server = http.createServer((req, res) => {
    res.write('Hello, World!');
    res.end();
});
server.listen(3000, () => console.log('Server running...'));
```

(d) OS

Provides operating system-related utility functions.

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

```
console.log('Platform:', os.platform());
console.log('CPU Architecture:', os.arch());
```

---

2. User-defined Modules

(a) Exporting from a Module

Create a file `math.js`:

```
function add(a, b) {
    return a + b;
}
function subtract(a, b) {
    return a - b;
}
module.exports = { add, subtract };
```

1. Import and use it in another file `app.js`:

```
const math = require('./math');
console.log(math.add(5, 3)); // 8
console.log(math.subtract(5, 3)); // 2
```

2. **(b) Exporting a Single Function or Value**

File `greet.js`:

```
module.exports = function (name) {
    return `Hello, ${name}!`;
};
```

File `app.js`:

```
const greet = require('./greet');
console.log(greet('Alice')); // Hello, Alice!
```

---

3. Third-party Modules

Third-party modules are installed via **npm**. Example:

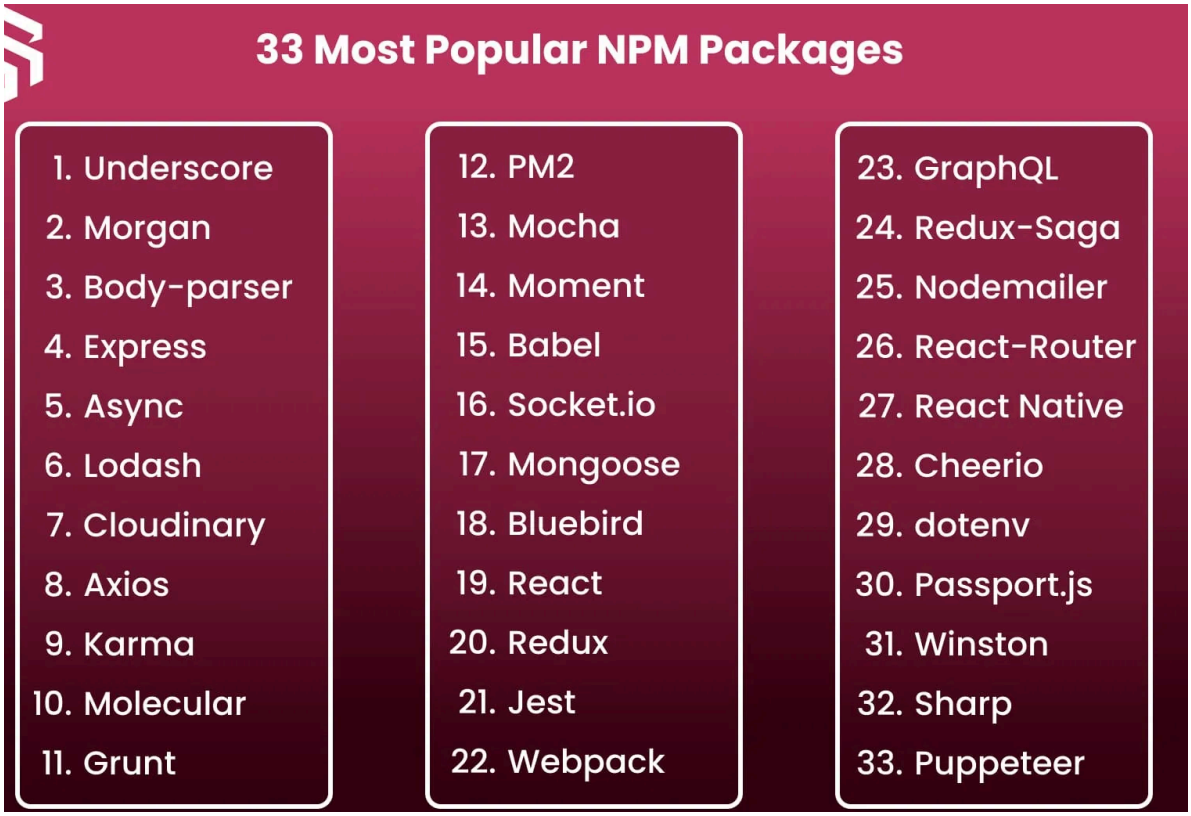
Installing a Module:

```
npm install lodash
```

Using the Module:

File `app.js`:

```
const _ = require('lodash');
const numbers = [10, 5, 8, 3];
console.log(_.sortBy(numbers)); // [3, 5, 8, 10]
```



---

4. Module Loading Mechanism

When you use `require`, Node.js follows a specific algorithm to locate the module:

1. **File or Directory**:
    - If the path starts with `./` or `/`, Node.js looks in the specified location.
    - Without a path, it assumes a core or third-party module.
2. **Order of Resolution**:
    - Core modules (e.g., `fs`).
    - File modules (`./filename.js`).
    - Third-party modules in `node_modules`.
3. **Caching**:

- Once a module is loaded, it is cached. Subsequent `require` calls return the cached version.

---

5. Advanced Module Concepts

(a) Using `exports`

You can use `exports` as a shorthand for `module.exports`.

```
exports.sayHello = function () {
    return 'Hello!';
};
```

(b) Circular Dependencies

When two modules depend on each other, Node.js resolves one fully before returning the partial exports of the other. Example:

`a.js`:

```
console.log('a.js loaded');
const b = require('./b');
module.exports = 'This is module A';
```

`b.js`:

```
console.log('b.js loaded');
const a = require('./a');
module.exports = 'This is module B';
```

(c) Dynamic Imports (ES Modules)

For dynamic, asynchronous imports:

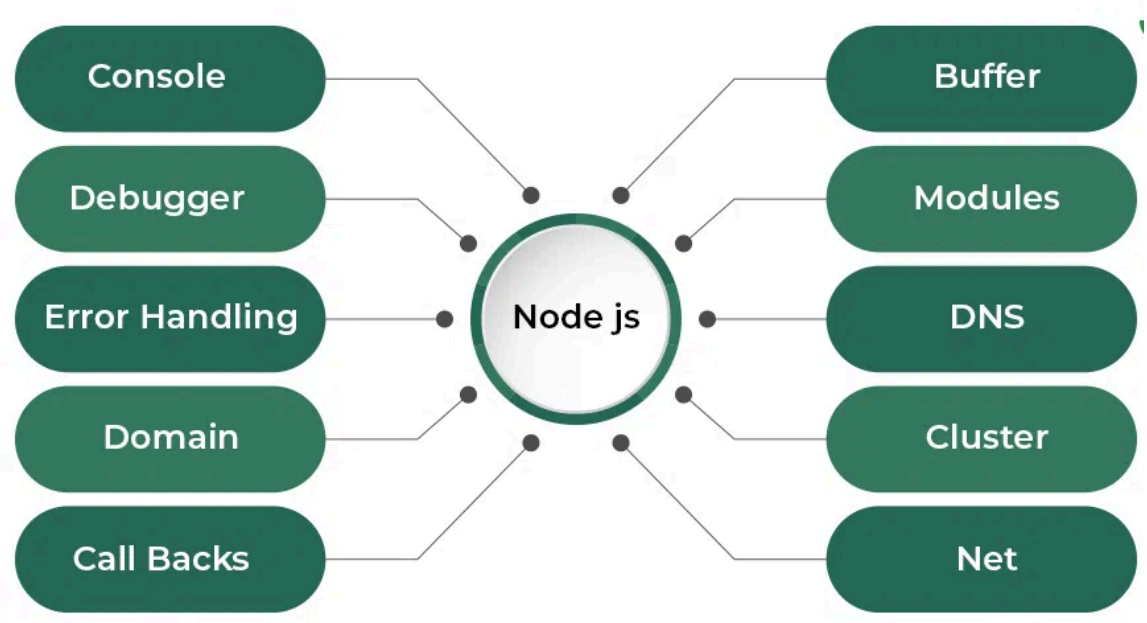
```
(async () => {
    const math = await import('./math.js');
    console.log(math.add(2, 3));
})();
```

Key Takeaways

1. Node.js modules are the backbone of modular programming.
2. Use **built-in modules** for common functionalities.
3. Write **user-defined modules** to keep your code clean and reusable.
4. Leverage **third-party modules** via npm for additional capabilities.
5. Understand how Node.js resolves, loads, and caches modules.

Chapter 3: Core Node.js Concepts

In this chapter, we'll explore the core concepts of Node.js that make it unique and powerful for building server-side applications.



---

1. Asynchronous Programming

Node.js is built on **asynchronous, non-blocking I/O**, enabling it to handle multiple requests concurrently without waiting for one to complete.

(a) Blocking Code Example:

```
const fs = require('fs');
const data = fs.readFileSync('file.txt', 'utf8'); // Synchronous
console.log(data); // Blocks until file is read
console.log('After Reading File');
```

- The program halts until `file.txt` is read.

(b) Non-blocking Code Example:

```
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => { // Asynchronous
    if (err) throw err;
    console.log(data);
});
console.log('After Reading File');
```

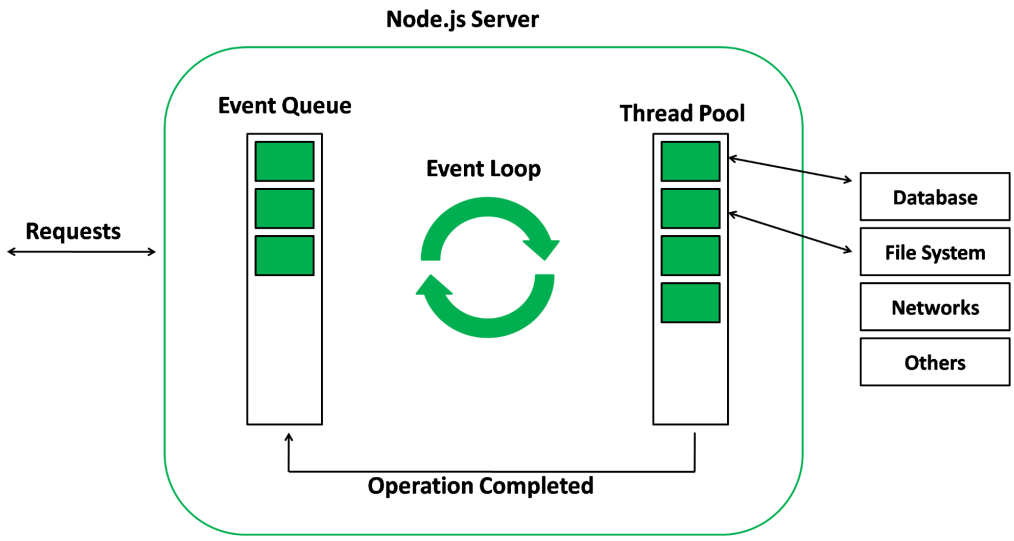
Output:

```
After Reading File
[Contents of file.txt]
```

---

2. The Event Loop

The **event loop** is the heart of Node.js, managing asynchronous tasks.



How It Works:

1. **Timers Phase:** Executes callbacks scheduled by `setTimeout` and `setInterval`.
2. **I/O Callbacks Phase:** Processes completed I/O operations.
3. **Idle/Prepare Phase:** Used internally by Node.js.
4. **Poll Phase:** Retrieves new I/O events.
5. **Check Phase:** Executes `setImmediate` callbacks.
6. **Close Callbacks Phase:** Executes `close` events (e.g., `socket.close`).

Example with `setTimeout` and `setImmediate`:

```
setTimeout(() => console.log('setTimeout'), 0);
setImmediate(() => console.log('setImmediate'));
console.log('Start');
```

Output:

Arduino code:

```
Start
setImmediate
setTimeout
```

---

3. Callbacks

A **callback** is a function passed as an argument to another function, executed after the main function is complete.

Example:

```
function fetchData(callback) {
    setTimeout(() => {
        callback('Data retrieved');
    }, 1000);
}

fetchData((message) => {
    console.log(message);
});
```

---

4. Promises

A **promise** represents a value that will be available in the future.

(a) Creating a Promise:

```
const myPromise = new Promise((resolve, reject) => {
    const success = true;
    if (success) resolve('Promise resolved');
    else reject('Promise rejected');
});

myPromise
    .then((message) => console.log(message))
    .catch((error) => console.error(error));
```

(b) Using `async` and `await`:

```
async function fetchData() {
    return 'Data fetched';
}

(async () => {
    const data = await fetchData();
    console.log(data);
})();
```

---

5. Streams

Streams handle data chunks efficiently, making them ideal for large files or real-time data.

Types of Streams:

1. **Readable Streams:** For reading data (e.g., `fs.createReadStream`).
2. **Writable Streams:** For writing data (e.g., `fs.createWriteStream`).
3. **Duplex Streams:** Both readable and writable.
4. **Transform Streams:** Modify data as it's read or written.

Example: Readable Stream

```
const fs = require('fs');
const stream = fs.createReadStream('file.txt', 'utf8');

stream.on('data', (chunk) => {
    console.log('New Chunk:', chunk);
});

stream.on('end', () => console.log('File read complete.'));
```

---

6. Buffer

A **Buffer** is a temporary storage area for raw binary data.

Example:

```
const buffer = Buffer.from('Hello, World!');
console.log(buffer); // <Buffer 48 65 6c 6c 6f 2c 20 57 6f 72 6c
64 21>
console.log(buffer.toString()); // Hello, World!
```

---

7. The `process` Object

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

(a) Accessing Environment Variables:

```
console.log(process.env.NODE_ENV); // e.g., 'development'
```

(b) Exiting the Process:

```
process.exit(0); // 0 indicates success
```

(c) Reading Command-Line Arguments:

```
console.log(process.argv); // Array of command-line arguments
```

---

8. Error Handling

Proper error handling ensures your application remains robust.

(a) Handling Errors with Callbacks:

```
fs.readFile('nonexistent.txt', 'utf8', (err, data) => {
    if (err) {
        console.error('Error occurred:', err.message);
        return;
    }
```

```
    console.log(data);
});
```

(b) Using `try...catch` with Promises:

```
async function fetchData() {
    try {
        const data = await someAsyncFunction();
        console.log(data);
    } catch (err) {
        console.error('Error:', err.message);
    }
}
```

---

9. Events and the EventEmitter

Node.js has an **EventEmitter** class to handle custom events.

Example:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('greet', (name) => {
    console.log(`Hello, ${name}!`);
});
```

```
emitter.emit('greet', 'Alice');
```

---

10. Timers

Node.js provides timer functions like `setTimeout`, `setInterval`, and `setImmediate`.

Example:

```
setTimeout(() => console.log('Timeout executed'), 1000); // Executes after 1 second
setInterval(() => console.log('Interval executed'), 2000); // Repeats every 2 seconds
```

Key Takeaways

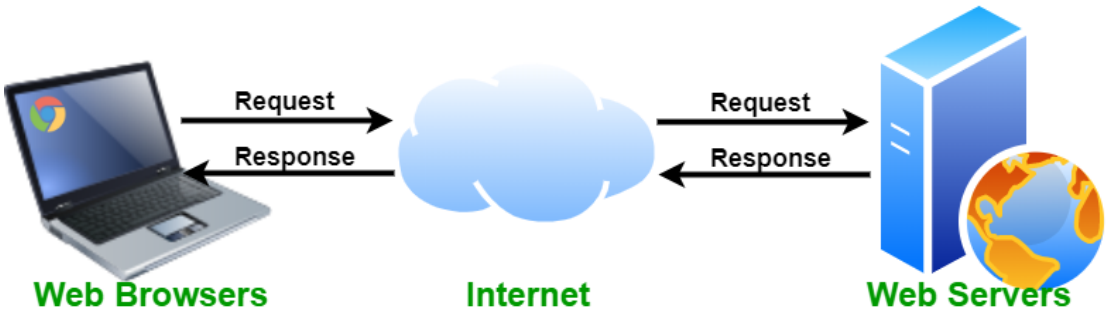
- Understand the asynchronous nature of Node.js.
- Learn how the event loop handles non-blocking I/O.
- Use callbacks, promises, and `async/await` for handling asynchronous tasks.
- Explore streams and buffers for efficient data handling.
- Leverage the `process` object for runtime information.
- Utilize the EventEmitter for event-driven programming.

Chapter 4: Building a Web Server with Node.js

In this chapter, we'll learn how to build a web server using Node.js. You'll understand how to handle HTTP requests, send responses, and build the foundation for creating web applications.

---

1. What Is an HTTP Web Server?



An **HTTP web server** is software that understands **HTTP** (the protocol used by browsers to view webpages) and **URLs** (web addresses). It stores websites and delivers their content to the end user's device. so,

A **web server**:

- Listens for incoming HTTP requests on a specific port.
- Processes the requests (e.g., reading data, querying a database).
- Sends back an HTTP response (e.g., HTML, JSON, files).

Node.js includes the `http` module for creating web servers.

---

2. Setting Up a Basic HTTP Server

Example:

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

// Create a server
const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/plain' }); // Set response headers
    res.write('Hello, World!'); // Write response body
    res.end(); // End the response
});

// Start the server
const PORT = 3000;
server.listen(PORT, () => {
    console.log(`Server running on http://localhost:${PORT}`);
});
```

How It Works:

1. **`http.createServer`** creates the server.
2. The callback `(req, res)` handles incoming requests.
    - `req`: Request object (client's input).
    - `res`: Response object (server's output).
3. `res.writeHead` sets the status code and headers.

4. `res.end` ends the response and sends it to the client.

---

3. Handling Requests

The `req` object contains information about the client's request:

- **URL**: The path requested by the client.
- **Method**: HTTP method (`GET`, `POST`, etc.).
- **Headers**: Metadata about the request.

Example:

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

const server = http.createServer((req, res) => {
    console.log(`Request URL: ${req.url}`);
    console.log(`Request Method: ${req.method}`);
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Request received');
});

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

---

4. Routing

Routing directs requests to different parts of the application based on the URL.

Example:

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

const server = http.createServer((req, res) => {
    if (req.url === '/' && req.method === 'GET') {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end('<h1>Welcome to the Home Page</h1>');
    } else if (req.url === '/about' && req.method === 'GET') {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end('<h1>About Us</h1>');
    } else {
        res.writeHead(404, { 'Content-Type': 'text/html' });
        res.end('<h1>404 - Page Not Found</h1>');
    }
});

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

---

5. Serving Static Files

To serve static files like HTML, CSS, or images, use the `fs` module.

Example:

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

const server = http.createServer((req, res) => {
    const filePath = req.url === '/' ? './index.html' :
`.${req.url}`;
    const ext = path.extname(filePath);

    let contentType = 'text/html';
    if (ext === '.css') contentType = 'text/css';
    if (ext === '.js') contentType = 'application/javascript';

    fs.readFile(filePath, (err, content) => {
        if (err) {
            res.writeHead(404, { 'Content-Type': 'text/html' });
            res.end('<h1>404 - File Not Found</h1>');
        } else {
            res.writeHead(200, { 'Content-Type': contentType });
            res.end(content);
        }
    });
});

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

---

6. Handling POST Requests

To handle POST requests, you need to capture data sent by the client.

Example:

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

const server = http.createServer((req, res) => {
    if (req.method === 'POST' && req.url === '/submit') {
        let body = '';

        req.on('data', (chunk) => {
            body += chunk.toString(); // Accumulate data chunks
        });

        req.on('end', () => {
            console.log('Data received:', body);
            res.writeHead(200, { 'Content-Type': 'text/plain'
});
            res.end('Data received');
        });
    } else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('Not Found');
    }
});
```

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

---

7. Creating a JSON API

To send and receive JSON data, use the `JSON.stringify` and `JSON.parse` methods.

Example:

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

const server = http.createServer((req, res) => {
    if (req.method === 'GET' && req.url === '/api') {
        const data = { message: 'Hello, API!' };
        res.writeHead(200, { 'Content-Type': 'application/json'
});
        res.end(JSON.stringify(data));
    } else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('Not Found');
    }
});

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

```
});
```

---

8. Middleware-like Functionality

Middleware functions are commonly used in frameworks like Express, but you can create similar functionality in raw Node.js.

Example:

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

const middleware = (req, res, next) => {
    console.log(`Request URL: ${req.url}`);
    console.log(`Request Method: ${req.method}`);
    next(); // Call the next function
};

const server = http.createServer((req, res) => {
    middleware(req, res, () => {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Middleware executed');
    });
});

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

```

```

Key Takeaways

1. Node.js allows you to create HTTP servers using the `http` module.
2. Routing can be implemented to handle different endpoints.
3. Use the `fs` module to serve static files.
4. Handle POST requests by capturing data chunks from the request body.
5. Build JSON APIs by sending structured data using `JSON.stringify`.

Chapter 5: Working with the File System

In this chapter, we'll explore the **Node.js File System (fs) module**, which allows you to interact with files and directories. You'll learn how to read, write, update, delete files, and manage directories.

---

1. The `fs` Module

The `fs` module in Node.js provides both **synchronous** and **asynchronous** methods for file and directory operations.

Importing the Module:

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

---

2. Reading Files

(a) Asynchronous Reading:

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

- **Parameters:**
    - `example.txt`: File path.
    - `utf8`: Encoding (optional).
    - Callback: Handles the file data or error.

(b) Synchronous Reading:

```
const data = fs.readFileSync('example.txt', 'utf8');
console.log(data);
```

- Blocks the program until the file is read.

---

3. Writing Files

(a) Asynchronous Writing:

```
fs.writeFile('example.txt', 'Hello, Node.js!', (err) => {
    if (err) throw err;
    console.log('File written successfully');
```

```
});
```

(b) Synchronous Writing:

```
fs.writeFileSync('example.txt', 'Hello, Node.js!');
console.log('File written successfully');
```

Append to a File:

```
fs.appendFile('example.txt', '\nAppended Text', (err) => {
    if (err) throw err;
    console.log('Text appended successfully');
});
```

---

4. Deleting Files

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

---

5. File Information

The `fs.stat` method retrieves file details.

Example:

```
fs.stat('example.txt', (err, stats) => {
    if (err) throw err;
    console.log('Is File:', stats.isFile());
    console.log('Is Directory:', stats.isDirectory());
    console.log('Size:', stats.size, 'bytes');
});
```

---

6. Working with Directories

Create a Directory:

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

Read a Directory:

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

Delete a Directory:

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

```
    if (err) throw err;
    console.log('Directory deleted');
});
```

---

7. File Streams

Streams allow you to process large files efficiently.

(a) Readable Stream:

```
const stream = fs.createReadStream('largefile.txt', 'utf8');
stream.on('data', (chunk) => {
    console.log('New Chunk:', chunk);
});
stream.on('end', () => console.log('Finished reading'));
```

(b) Writable Stream:

```
const writeStream = fs.createWriteStream('output.txt');
writeStream.write('Hello, World!\n');
writeStream.write('Stream writing!');
writeStream.end(() => console.log('Write complete'));
```

(c) Pipe Streams:

Pipes connect readable streams to writable streams.

```
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');
readStream.pipe(writeStream);
console.log('File copied successfully');
```

---

8. Watching Files and Directories

Example:

```
fs.watch('example.txt', (eventType, filename) => {
    console.log(`Event: ${eventType}`);
    console.log(`File: ${filename}`);
});
```

---

9. Error Handling

Always handle errors when working with the file system to avoid crashes.

Example:

```
fs.readFile('nonexistent.txt', 'utf8', (err, data) => {
    if (err) {
        console.error('Error:', err.message);
        return;
    }
    console.log(data);
```

```
});
```

---

10. Practical Example: File Manager

Here's a basic file manager that reads, writes, and deletes files based on user input.

Example:

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

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
});

rl.question('Enter a command (read/write/delete): ', (command)
=> {
    if (command === 'read') {
        rl.question('Enter file name: ', (filename) => {
            fs.readFile(filename, 'utf8', (err, data) => {
                if (err) console.error('Error:', err.message);
                else console.log(data);
                rl.close();
            });
        });
```

```
    } else if (command === 'write') {
        rl.question('Enter file name: ', (filename) => {
            rl.question('Enter content: ', (content) => {
                fs.writeFile(filename, content, (err) => {
                    if (err) console.error('Error:',
err.message);
                    else console.log('File written
successfully');
                    rl.close();
                });
            });
        });
    } else if (command === 'delete') {
        rl.question('Enter file name: ', (filename) => {
            fs.unlink(filename, (err) => {
                if (err) console.error('Error:', err.message);
                else console.log('File deleted successfully');
                rl.close();
            });
        });
    } else {
        console.log('Invalid command');
        rl.close();
    }
});
```

Key Takeaways

1. The `fs` module is powerful for file and directory operations.
2. Choose asynchronous methods for non-blocking operations.
3. Use streams for large files to improve performance.
4. Always handle errors gracefully to avoid application crashes.

Chapter 6: Working with Databases in Node.js

In this chapter, you'll learn how to interact with databases in Node.js. We'll cover connecting to databases, running queries, and managing data efficiently. We'll explore SQL (e.g., MySQL, PostgreSQL) and NoSQL (e.g., MongoDB) databases.

---

1. Choosing a Database

- **SQL (Relational):** MySQL, PostgreSQL, SQLite.
    - Structured, tabular data with relationships.
    - Uses SQL for queries.
- **NoSQL (Non-relational):** MongoDB, Cassandra.
    - Flexible, schema-less, JSON-like data.
    - Suitable for hierarchical or unstructured data.

---

2. Setting Up a Database

Before using a database, ensure it’s installed and running on your system or accessible through a cloud provider.

---

3. Working with SQL Databases

Example: MySQL

(a) Install the MySQL Library:

```
npm install mysql
```

(b) Connect to MySQL:

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

// Create connection
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'password',
    database: 'testdb',
});

// Connect to database
```

```
connection.connect((err) => {
    if (err) throw err;
    console.log('Connected to MySQL');
});
```

(c) Perform Queries:

- **Create a Table:**

```
const sql = `CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    email VARCHAR(255)
)`;
connection.query(sql, (err, result) => {
    if (err) throw err;
    console.log('Table created');
});
```

- **Insert Data:**

```
const sql = `INSERT INTO users (name, email) VALUES ('Alice',
'alice@example.com')`;
connection.query(sql, (err, result) => {
    if (err) throw err;
    console.log('Data inserted', result);
});
```

- **Fetch Data:**

```
connection.query('SELECT * FROM users', (err, results) => {
    if (err) throw err;
    console.log('Data fetched:', results);
});
```

- **Close the Connection:**

```
connection.end((err) => {
    if (err) throw err;
    console.log('Connection closed');
});
```

---

4. Working with NoSQL Databases

Example: MongoDB

(a) Install MongoDB Library:

```
npm install mongodb
```

(b) Connect to MongoDB:

```
const { MongoClient } = require('mongodb');
```

```
// Connection URL
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

// Database and Collection
const dbName = 'testdb';
const collectionName = 'users';

(async () => {
    try {
        await client.connect();
        console.log('Connected to MongoDB');
        const db = client.db(dbName);
        const collection = db.collection(collectionName);

        // Perform operations here

    } catch (err) {
        console.error('Error:', err.message);
    } finally {
        await client.close();
        console.log('Connection closed');
    }
})();
```

(c) Perform Operations:

- **Insert Documents:**

```
await collection.insertOne({ name: ‘Mohammed’, email:
‘moh@emmerisve.com' });
console.log('Document inserted');
```

- **Find Documents:**

```
const users = await collection.find({}).toArray();
console.log('Documents found:', users);
```

- **Update Documents:**

```
await collection.updateOne({ name: ‘mohammed’}, { $set: { email:
newmoh@emmersive.com' } });
console.log('Document updated');
```

- **Delete Documents:**

```
await collection.deleteOne({ name: 'Bob' });
console.log('Document deleted');
```

---

5. Using an ORM (Object-Relational Mapping)

ORMs simplify database operations by abstracting SQL or NoSQL syntax.

Example: Sequelize (for SQL)

```
npm install sequelize mysql2
```

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('testdb', 'root', 'password', {
    host: 'localhost',
    dialect: 'mysql',
});

// Define a Model
const User = sequelize.define('User', {
    name: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    email: {
        type: DataTypes.STRING,
        allowNull: false,
    },
});

// Sync and Use the Model
(async () => {
    await sequelize.sync();
    console.log('Database synchronized');

    // Insert a user
    await User.create({ name: ‘mohammed’, email:
```

```
moha@example.com' });
    console.log('User created');

    // Fetch users
    const users = await User.findAll();
    console.log('Users:', users);
})();
```

Example: Mongoose (for MongoDB)

```
npm install mongoose
```

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

// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/testdb', {
useNewUrlParser: true, useUnifiedTopology: true })
    .then(() => console.log('Connected to MongoDB'))
    .catch((err) => console.error('Connection error:', err));

// Define a Schema
const userSchema = new mongoose.Schema({
    name: String,
    email: String,
});

// Create a Model
```

```
const User = mongoose.model('User', userSchema);

// Perform Operations
(async () => {
    const user = new User({ name: 'Diana', email:
'diana@example.com' });
    await user.save();
    console.log('User saved');

    const users = await User.find();
    console.log('Users:', users);
})();
```

---

6. Database Best Practices

1. **Use Environment Variables:** Store sensitive credentials in `.env` files using the `dotenv` package.
2. **Sanitize Inputs:** Prevent SQL injection by using prepared statements or ORM methods.
3. **Index Data:** Index fields that are queried frequently for faster retrieval.
4. **Backup Data:** Regularly back up your database to prevent data loss.
5. **Monitor Performance:** Use monitoring tools for database health and query optimization.

Key Takeaways

1. SQL and NoSQL databases serve different use cases—choose based on your application needs.
2. Node.js provides libraries for seamless database interactions.
3. ORMs like Sequelize and Mongoose simplify database operations but may introduce abstraction overhead.
4. Always prioritize security, scalability, and efficiency when working with databases.

Chapter 7: Understanding Asynchronous Programming in Node.js

Asynchronous programming is a cornerstone of Node.js, enabling it to handle multiple tasks simultaneously without blocking the execution of code. In this chapter, we'll dive deep into how Node.js handles asynchronous operations, exploring callbacks, promises, async/await, and the Event Loop.

---

1. Why Asynchronous Programming?

- Node.js operates on a **single-threaded, non-blocking** architecture.
- Asynchronous programming ensures tasks like file I/O, network requests, and database queries don't block the main thread, allowing other operations to continue.

---

2. The Event Loop

The **Event Loop** is the mechanism that allows Node.js to perform non-blocking I/O operations by offloading tasks to the system kernel or worker threads.

Phases of the Event Loop:

1. **Timers:** Executes `setTimeout` and `setInterval` callbacks.
2. **I/O Callbacks:** Executes callbacks for deferred I/O operations.
3. **Idle/Prepare:** Used internally.
4. **Poll:** Retrieves new I/O events and executes their callbacks.
5. **Check:** Executes `setImmediate` callbacks.
6. **Close Callbacks:** Handles closed events (e.g., `socket.on('close')`).

---

3. Callbacks

A **callback** is a function passed as an argument to another function, invoked after the completion of an asynchronous task.

Example:

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

fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) return console.error('Error:', err);
    console.log('File content:', data);
});
```

Problem with Callbacks:

- **Callback Hell:** Nested callbacks make code hard to read and maintain.

Example of Callback Hell:

```
doSomething((err, result) => {
    if (err) throw err;
    doSomethingElse(result, (err, newResult) => {
        if (err) throw err;
        doAnotherThing(newResult, (err, finalResult) => {
            if (err) throw err;
            console.log(finalResult);
        });
    });
});
```

---

4. Promises

A **Promise** represents a value that will be available in the future. It has three states:

1. **Pending**
2. **Resolved (fulfilled)**
3. **Rejected**

Creating a Promise:

```
const myPromise = new Promise((resolve, reject) => {
    const success = true; // Simulate condition
    if (success) resolve('Promise resolved!');
    else reject('Promise rejected!');
```

```
});

myPromise
    .then((message) => console.log(message))
    .catch((error) => console.error(error));
```

Chaining Promises:

```
fetchData()
    .then((data) => processData(data))
    .then((processedData) => saveData(processedData))
    .catch((error) => console.error('Error:', error));
```

---

5. Async/Await

Introduced in ES2017, **async/await** simplifies working with promises and makes asynchronous code look synchronous.

Example:

```
const fetchData = async () => {
    try {
        const data = await someAsyncFunction();
        console.log(data);
    } catch (error) {
        console.error('Error:', error);
```

```
    }
};

fetchData();
```

Key Points:

- `async` makes a function return a promise.
- `await` pauses execution until the promise resolves or rejects.

---

6. Handling Asynchronous Operations

(a) `setTimeout` and `setInterval`

```
setTimeout(() => console.log('Executed after 2 seconds'), 2000);

let count = 0;
const intervalId = setInterval(() => {
    count += 1;
    console.log(`Interval count: ${count}`);
    if (count === 5) clearInterval(intervalId);
}, 1000);
```

(b) `setImmediate`

```
setImmediate(() => console.log('Executed immediately after the current event loop phase'));
```

---

7. Practical Examples

Example 1: Simulating API Calls

```
const fakeAPI = (url) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (url === 'validURL') resolve('Data fetched successfully');
            else reject('Invalid URL');
        }, 1000);
    });
};

(async () => {
    try {
        const data = await fakeAPI('validURL');
        console.log(data);
    } catch (error) {
        console.error('Error:', error);
    }
})();
```

Example 2: File Reading with Promises

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

(async () => {
    try {
        const data = await fs.readFile('example.txt', 'utf8');
        console.log('File content:', data);
    } catch (err) {
        console.error('Error:', err);
    }
})();
```

Example 3: Combining Async Tasks

```
const task1 = () => new Promise((resolve) => setTimeout(() =>
resolve('Task 1 completed'), 1000));
const task2 = () => new Promise((resolve) => setTimeout(() =>
resolve('Task 2 completed'), 500));

(async () => {
    console.log(await task1());
    console.log(await task2());
})();
```

8. Common Mistakes in Async Code

1. **Uncaught Errors:** Always use `.catch()` or `try-catch` to handle promise rejections.
2. **Blocking Code:** Avoid mixing synchronous and asynchronous code that may block the event loop.

Overuse of `await`: Use `Promise.all` to handle multiple promises in parallel.

```
const [result1, result2] = await Promise.all([task1(), task2()]);
```

---

9. Best Practices for Async Programming

1. Prefer `async/await` for readability.
2. Always handle errors with `.catch()` or `try-catch`.
3. Use `Promise.all` for parallel execution.
4. Avoid blocking the event loop with heavy synchronous computations.
5. Test async code thoroughly to catch edge cases.

Key Takeaways

1. Node.js thrives on asynchronous programming to handle high-concurrency tasks efficiently.
2. The Event Loop is crucial to understanding Node.js's non-blocking architecture.

3. Use promises and `async/await` to write clean, maintainable asynchronous code.
4. Manage errors gracefully to ensure robustness.

Chapter 8: Networking and HTTP in Node.js

Node.js is widely used for building networking applications and HTTP servers due to its asynchronous, event-driven architecture. In this chapter, we'll explore networking basics, creating HTTP servers, routing, and working with APIs.

---

1. Understanding Networking in Node.js

Node.js provides the `net` module for lower-level networking and the `http` module for building web servers. Common use cases include:

- Creating TCP servers and clients.
- Building HTTP/HTTPS servers.
- Handling requests and responses.
- Communicating with APIs.

---

2. Building a Basic HTTP Server

Example:

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

```
// Create a server
const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, World!');
});

// Listen on a port
const PORT = 3000;
server.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}/`);
});
```

---

3. Handling Requests and Responses

Example:

```
const server = http.createServer((req, res) => {
    if (req.url === '/' && req.method === 'GET') {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end('<h1>Welcome to the Homepage!</h1>');
    } else if (req.url === '/about' && req.method === 'GET') {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end('<h1>About Us</h1>');
    } else {
        res.writeHead(404, { 'Content-Type': 'text/html' });
        res.end('<h1>404 - Page Not Found</h1>');
```

```
    }
});
```

---

4. Parsing URL Parameters

Using `url` Module:

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

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

    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(`Query Parameters: ${JSON.stringify(query)}`);
});
```

Example URL:

```
http://localhost:3000/?name=John&age=30
```

Output:

```
Query Parameters: {"name":"John","age":"30"}
```

---

5. Serving Static Files

Use the `fs` module to serve static files.

Example:

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

const server = http.createServer((req, res) => {
    const filePath = path.join(__dirname, 'public', req.url === '/' ? 'index.html' : req.url);
    const ext = path.extname(filePath);

    let contentType = 'text/html';
    if (ext === '.css') contentType = 'text/css';
    else if (ext === '.js') contentType = 'application/javascript';

    fs.readFile(filePath, (err, content) => {
        if (err) {
            res.writeHead(404, { 'Content-Type': 'text/html' });
            res.end('<h1>File Not Found</h1>');
        } else {
            res.writeHead(200, { 'Content-Type': contentType });
            res.end(content);
        }
```

```
    });
});

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

---

6. Building a Simple REST API

Example:

```
const data = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
];

const server = http.createServer((req, res) => {
    if (req.url === '/api/users' && req.method === 'GET') {
        res.writeHead(200, { 'Content-Type': 'application/json'
});
        res.end(JSON.stringify(data));
    } else if (req.url === '/api/users' && req.method ===
'POST') {
        let body = '';
        req.on('data', (chunk) => (body += chunk));
        req.on('end', () => {
```

```
            const newUser = JSON.parse(body);
            data.push(newUser);
            res.writeHead(201, { 'Content-Type':
'application/json' });
            res.end(JSON.stringify(newUser));
        });
    } else {
        res.writeHead(404, { 'Content-Type': 'application/json'
});
        res.end(JSON.stringify({ message: 'Not Found' }));
    }
});

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

---

7. Using HTTPS

For secure communication, use the `https` module.

Example:

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

```
// Load SSL certificates
const options = {
    key: fs.readFileSync('key.pem'),
    cert: fs.readFileSync('cert.pem'),
};

https.createServer(options, (req, res) => {
    res.writeHead(200);
    res.end('Hello, secure world!');
}).listen(443, () => {
    console.log('HTTPS server running on port 443');
});
```

---

8. Third-Party Modules: Express.js

While the built-in `http` module is powerful, frameworks like **Express.js** simplify server creation.

Example:

```
npm install express
```

Then :

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

```
const app = express();

app.get('/', (req, res) => res.send('Welcome to Express!'));
app.get('/api/users', (req, res) => res.json([{ id: 1, name:
'Alice' }]));

const PORT = 3000;
app.listen(PORT, () => console.log(`Express server running on
port ${PORT}`));
```

---

9. Networking with the `net` Module

Use the `net` module for lower-level TCP/UDP communication.

TCP Server Example:

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

const server = net.createServer((socket) => {
    console.log('Client connected');
    socket.write('Welcome to the TCP server!\n');

    socket.on('data', (data) => {
        console.log('Received:', data.toString());
        socket.write(`Echo: ${data}`);
```

```
    });

    socket.on('end', () => console.log('Client disconnected'));
});

server.listen(8080, () => console.log('TCP server running on
port 8080'));
```

TCP Client Example:

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

const client = net.createConnection({ port: 8080 }, () => {
    console.log('Connected to server');
    client.write('Hello, server!');
});

client.on('data', (data) => {
    console.log('Received:', data.toString());
    client.end();
});

client.on('end', () => console.log('Disconnected from server'));
```

---

10. Best Practices

1. **Use Frameworks:** Simplify development with frameworks like Express.js.
2. **Validate Input:** Always sanitize and validate user inputs.
3. **Use HTTPS:** Secure sensitive data using HTTPS.
4. **Error Handling:** Gracefully handle errors to prevent server crashes.
5. **Load Balancing:** Use tools like Nginx or PM2 for high-traffic applications.

Key Takeaways

1. The `http` module allows you to build robust web servers.
2. For lower-level networking, the `net` module is useful for TCP/UDP applications.
3. Frameworks like Express.js simplify routing and middleware integration.
4. Secure communication is critical; use HTTPS wherever possible

Chapter 9: Databases and ORMs in Node.js

Databases are integral to backend applications, providing a way to persist and manage data. In this chapter, we'll explore how to integrate databases into a Node.js application, covering SQL and NoSQL options, and leveraging ORMs for productivity.

---

1. Overview of Databases

Types of Databases:

1. **Relational Databases (SQL):** Use structured tables with predefined schemas. Examples: MySQL, PostgreSQL, SQLite.
2. **NoSQL Databases:** Use flexible, schema-less designs. Examples: MongoDB, CouchDB, Redis.

---

2. Working with SQL Databases

Example: PostgreSQL

Installing PostgreSQL Driver:

```
npm install pg
```

Connecting to PostgreSQL:

```
const { Pool } = require('pg');

const pool = new Pool({
    user: 'your_username',
    host: 'localhost',
    database: 'your_database',
    password: 'your_password',
    port: 5432,
});

(async () => {
    try {
        const res = await pool.query('SELECT NOW()');
```

```
        console.log('Connected to PostgreSQL:', res.rows);
    } catch (err) {
        console.error('Error connecting to PostgreSQL:', err);
    } finally {
        pool.end();
    }
})();
```

Performing CRUD Operations:

- **Create:**

```
await pool.query('INSERT INTO users (name, email) VALUES ($1, $2)', ['John Doe', 'john@example.com']);
```

- **Read:**

```
const res = await pool.query('SELECT * FROM users');
console.log('Users:', res.rows);
```

- **Update:**

```
await pool.query('UPDATE users SET name = $1 WHERE email = $2', ['Jane Doe', 'john@example.com']);
```

- **Delete:**

```
await pool.query('DELETE FROM users WHERE email = $1',
['john@example.com']);
```

---

3. Working with NoSQL Databases

Example: MongoDB

Installing MongoDB Driver:

```
npm install mongodb
```

Connecting to MongoDB:

```
const { MongoClient } = require('mongodb');

const uri = 'mongodb://localhost:27017';
const client = new MongoClient(uri);

(async () => {
    try {
        await client.connect();
        console.log('Connected to MongoDB');
        const db = client.db('mydatabase');
        const users = db.collection('users');
```

```
        // CRUD operations here
    } catch (err) {
        console.error('Error connecting to MongoDB:', err);
    } finally {
        await client.close();
    }
})();
```

Performing CRUD Operations:

- **Create:**

```
await users.insertOne({ name: 'John Doe', email:
'john@example.com' });
```

- **Read:**

```
const user = await users.findOne({ email: 'john@example.com' });
console.log('User:', user);
```

- **Update:**

```
await users.updateOne({ email: 'john@example.com' }, { $set: {
name: 'Jane Doe' } });
```

- **Delete:**

```
await users.deleteOne({ email: 'john@example.com' });
```

---

4. Using ORMs

ORMs (Object-Relational Mappers) simplify database interactions by abstracting SQL/NoSQL queries into JavaScript methods.

Popular ORMs:

1. **SQL:** Sequelize, TypeORM
2. **NoSQL:** Mongoose (for MongoDB)

Using Sequelize (SQL ORM):

Installation:

```
npm install sequelize pg
```

Initialization:

```
const { Sequelize, DataTypes } = require('sequelize');

const sequelize = new Sequelize('database', 'username',
```

```
'password', {
    host: 'localhost',
    dialect: 'postgres',
});

const User = sequelize.define('User', {
    name: { type: DataTypes.STRING, allowNull: false },
    email: { type: DataTypes.STRING, unique: true },
});

// Syncing models
(async () => {
    try {
        await sequelize.authenticate();
        console.log('Connected to PostgreSQL using Sequelize');
        await sequelize.sync();
    } catch (err) {
        console.error('Error:', err);
    }
})();
```

CRUD Operations:

- **Create:**

```
await User.create({ name: 'John Doe', email: 'john@example.com'
});
```

- **Read:**

```
const users = await User.findAll();
console.log(users);
```

- **Update:**

```
await User.update({ name: 'Jane Doe' }, { where: { email:
'john@example.com' } });
```

- **Delete:**

```
await User.destroy({ where: { email: 'john@example.com' } });
```

Using Mongoose (NoSQL ORM):

Installation:

```
npm install mongoose
```

Initialization:

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

```
mongoose.connect('mongodb://localhost:27017/mydatabase', {
    useNewUrlParser: true,
    useUnifiedTopology: true,
});

const userSchema = new mongoose.Schema({
    name: String,
    email: { type: String, unique: true },
});

const User = mongoose.model('User', userSchema);

(async () => {
    try {
        console.log('Connected to MongoDB using Mongoose');
        // CRUD operations here
    } catch (err) {
        console.error('Error:', err);
    }
})();
```

CRUD Operations:

- **Create:**

```
await User.create({ name: 'John Doe', email: 'john@example.com'
```

```
});
```

- **Read:**

```
const user = await User.findOne({ email: 'john@example.com' });
console.log('User:', user);
```

- **Update:**

```
await User.updateOne({ email: 'john@example.com' }, { name:
'Jane Doe' });
```

- **Delete:**

```
await User.deleteOne({ email: 'john@example.com' });
```

---

5. Choosing Between SQL and NoSQL

Feature	SQL	NoSQL
Structure	Tables with schemas	Collections, flexible
Scaling	Vertical	Horizontal
Data Relations	Strong (joins)	Weak (embedded docs)
Use Cases	Structured, transactional	Flexible, large-scale

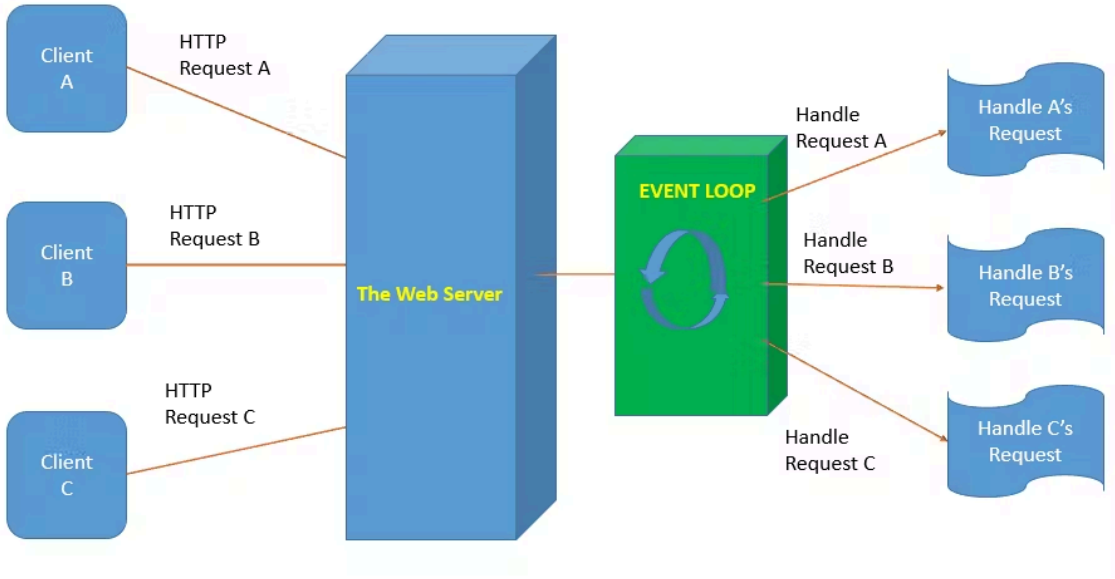
---

6. Best Practices

1. **Use Connection Pooling:** To manage resources efficiently.
2. **Environment Variables:** Store sensitive credentials securely (e.g., with `dotenv`).
3. **Validate Inputs:** Prevent SQL/NoSQL injection attacks.
4. **Backup Data Regularly:** Especially in production environments.

Chapter 10: Handling APIs and HTTP Requests in Node.js

In this chapter, we will explore how to handle external API calls and HTTP requests effectively in Node.js. APIs allow applications to communicate with external services, enabling features like fetching data from third-party services or integrating payment systems.

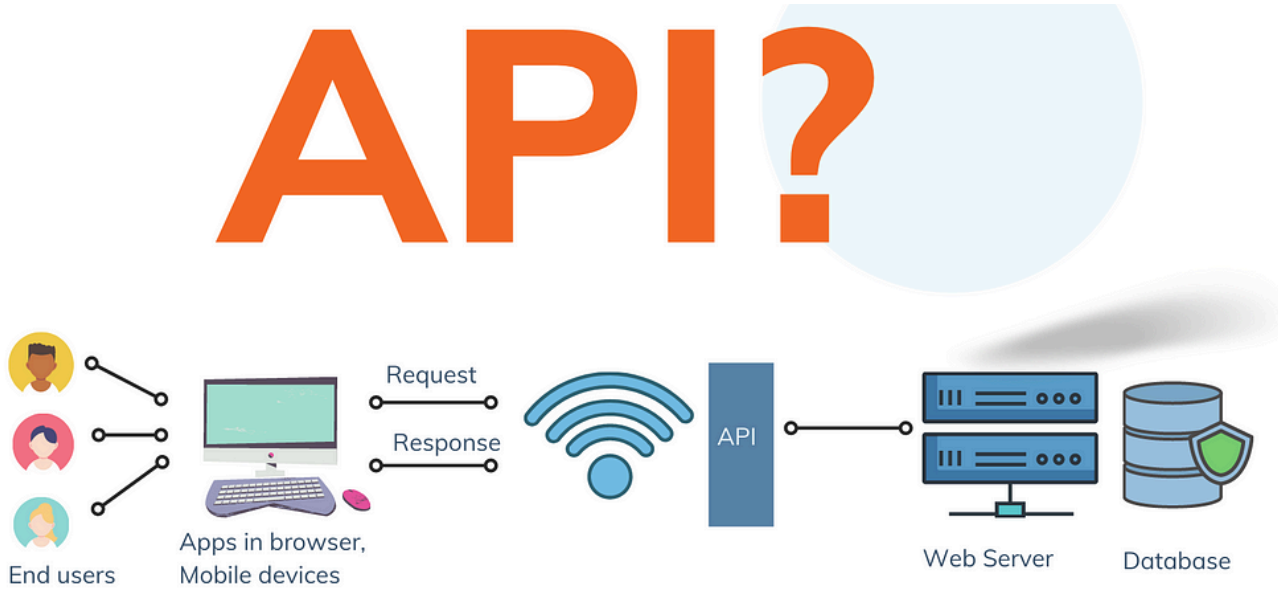


---

1. Understanding APIs and HTTP Requests

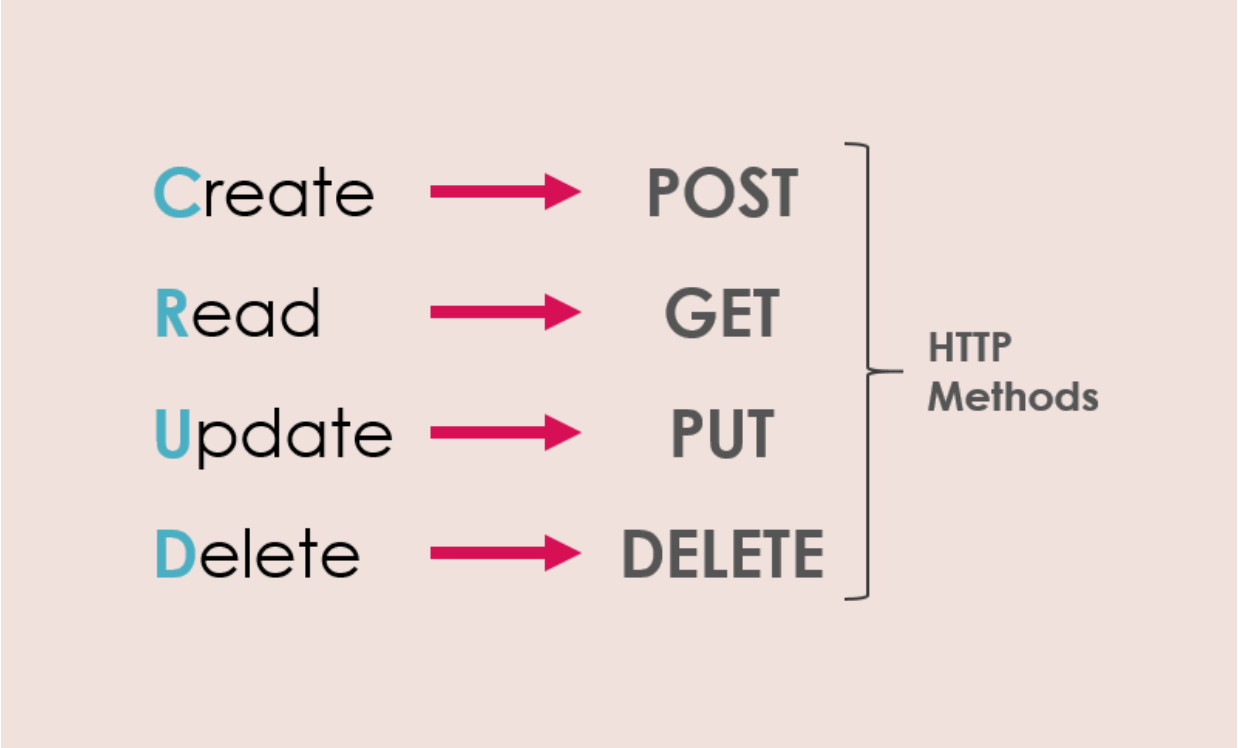
API Basics

- **API (Application Programming Interface):** A set of rules that allows communication between software components.
- **REST API:** A widely used style for building APIs that use HTTP methods like GET, POST, PUT, and DELETE.



HTTP Methods

- **GET:** Retrieve data.
- **POST:** Send new data.
- **PUT:** Update existing data.
- **DELETE:** Remove data.



---

2. Using `http` and `https` Modules

Node.js includes built-in modules to handle HTTP/HTTPS requests.

Example: Making an HTTP GET Request

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

http.get('http://jsonplaceholder.typicode.com/posts/1', (res) =>
{
    let data = '';

    res.on('data', (chunk) => {
```

```
        data += chunk;
    });

    res.on('end', () => {
        console.log(JSON.parse(data));
    });
}).on('error', (err) => {
    console.error('Error:', err.message);
});
```

---

3. Using Third-Party Libraries

Libraries like `axios` and `node-fetch` make HTTP requests simpler and more feature-rich.

Installing `axios`:

```
npm install axios
```

Example: Using `axios`

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

(async () => {
    try {
        const response = await
```

```
axios.get('https://jsonplaceholder.typicode.com/posts/1');
        console.log(response.data);
    } catch (err) {
        console.error('Error:', err.message);
    }
})();
```

---

4. Handling API Requests in a Node.js Server

Example: Proxying API Requests

You can create a route in your Node.js server that proxies a third-party API.

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

app.get('/api/posts/:id', async (req, res) => {
    try {
        const response = await
axios.get(`https://jsonplaceholder.typicode.com/posts/${req.para
ms.id}`);
        res.json(response.data);
    } catch (err) {
        res.status(500).json({ error: 'Failed to fetch data' });
    }
```

```
});

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

---

5. Sending POST Requests

Example: POST Request with `axios`

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

(async () => {
    try {
        const response = await
axios.post('https://jsonplaceholder.typicode.com/posts', {
            title: 'foo',
            body: 'bar',
            userId: 1,
        });
        console.log(response.data);
    } catch (err) {
        console.error('Error:', err.message);
    }
})();
```

---

6. Error Handling in API Calls

Always handle potential errors during API requests.

Example:

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

(async () => {
    try {
        const response = await
axios.get('https://jsonplaceholder.typicode.com/invalid-endpoint
');
        console.log(response.data);
    } catch (err) {
        if (err.response) {
            // Server responded with a status other than 2xx
            console.error('Error Response:',
err.response.status);
        } else if (err.request) {
            // Request was made but no response received
            console.error('No Response:', err.request);
        } else {
            // Other errors
            console.error('Error:', err.message);
        }
    }
})();
```

---

7. Calling APIs from Frontend

You can integrate APIs with your frontend using libraries like `axios` or `fetch`.

Example: Fetching Data in React

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';

function App() {
    const [post, setPost] = useState(null);

    useEffect(() => {
        const fetchPost = async () => {
            const response = await
axios.get('http://localhost:3000/api/posts/1');
            setPost(response.data);
        };

        fetchPost();
    }, []);

    return (
        <div>
            {post ? <h1>{post.title}</h1> : <p>Loading...</p>}
        </div>
    );
}
```

```
export default App;
```

---

8. Best Practices for Handling APIs

1. **Timeouts:** Set timeouts for requests to prevent the application from hanging.
2. **Rate Limiting:** Prevent abuse by limiting the number of API calls.
3. **Caching:** Use caching mechanisms for frequently accessed data.
4. **Authentication:** Secure your API endpoints with tokens (e.g., JWT, API keys).
5. **Pagination:** Use pagination for large datasets to improve performance.

---

9. Practical Example: Weather App

Create a Weather API Proxy

```
const express = require('express');
const axios = require('axios');
require('dotenv').config();

const app = express();
const WEATHER_API_URL =
'https://api.openweathermap.org/data/2.5/weather';
const WEATHER_API_KEY = process.env.WEATHER_API_KEY;

app.get('/api/weather', async (req, res) => {
```

```
    const { city } = req.query;
    if (!city) {
        return res.status(400).json({ error: 'City is required'
});
    }

    try {
        const response = await axios.get(WEATHER_API_URL, {
            params: {
                q: city,
                appid: WEATHER_API_KEY,
            },
        });
        res.json(response.data);
    } catch (err) {
        res.status(500).json({ error: 'Failed to fetch weather
data' });
    }
});

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

Test the Weather API:

Start the server:

```
node server.js
```

Access:

```
http://localhost:3000/api/weather?city=London
```

Chapter 11: Authentication and Authorization in Node.js

Authentication and authorization are crucial in securing applications. Authentication verifies user identity, while authorization determines user permissions.

In this chapter, we'll learn to:

1. Implement user authentication using **JWT (JSON Web Tokens)**.
2. Secure routes and resources.
3. Understand the difference between authentication and authorization.

---

1. Key Concepts

Authentication

- Verifies the user's identity using credentials (e.g., username and password).

Authorization

- Determines what actions or resources an authenticated user is allowed to access.

JWT (JSON Web Tokens)

- A compact, self-contained token used to securely transmit information between parties.
- **Structure:** Header, Payload, Signature.

---

2. Setup and Dependencies

Install Required Libraries

```
npm install bcrypt jsonwebtoken express-validator dotenv
```

---

3. User Authentication Example

We'll build a system where users can register, log in, and access protected routes.

Directory Structure

```
auth-example/
├── models/
│   └── user.js
├── routes/
│   └── authRoutes.js
├── config/
│   └── database.js
├── .env
├── server.js
```

Step 1: Define the User Model

models/user.js

```
const { DataTypes } = require('sequelize');
const sequelize = require('../config/database');
const bcrypt = require('bcrypt');

const User = sequelize.define('User', {
```

```
    username: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true,
    },
    password: {
        type: DataTypes.STRING,
        allowNull: false,
    },
});

// Hash password before saving
User.beforeCreate(async (user) => {
    const salt = await bcrypt.genSalt(10);
    user.password = await bcrypt.hash(user.password, salt);
});

module.exports = User;
```

Step 2: Create Authentication Routes

routes/authRoutes.js

```
const express = require('express');
const { User } = require('../models');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcrypt');
```

```
const { body, validationResult } = require('express-validator');
require('dotenv').config();

const router = express.Router();

// Register a new user
router.post(
    '/register',
    body('username').isString().notEmpty(),
    body('password').isLength({ min: 6 }),
    async (req, res) => {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array()
});
        }

        try {
            const { username, password } = req.body;
            const user = await User.create({ username, password
});
            res.status(201).json({ message: 'User registered',
user });
        } catch (err) {
            res.status(500).json({ error: err.message });
        }
    }
```

```
);

// Login a user
router.post(
    '/login',
    body('username').isString(),
    body('password').notEmpty(),
    async (req, res) => {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array()
});
        }

        try {
            const { username, password } = req.body;
            const user = await User.findOne({ where: { username
} });

            if (!user || !(await bcrypt.compare(password,
user.password))) {
                return res.status(401).json({ error: 'Invalid
credentials' });
            }

            const token = jwt.sign({ id: user.id, username:
user.username }, process.env.JWT_SECRET, {
```

```
                expiresIn: '1h',
            });

            res.json({ message: 'Login successful', token });
        } catch (err) {
            res.status(500).json({ error: err.message });
        }
    }
);

module.exports = router;
```

Step 3: Protect Routes

Middleware for Authentication

```
const jwt = require('jsonwebtoken');

const authenticateToken = (req, res, next) => {
    const token = req.headers['authorization'];

    if (!token) {
        return res.status(403).json({ error: 'No token provided'
});
    }

    jwt.verify(token.split(' ')[1], process.env.JWT_SECRET,
```

```
(err, user) => {
        if (err) {
            return res.status(401).json({ error: 'Invalid token'
});
        }
        req.user = user;
        next();
    });
};

module.exports = authenticateToken;
```

Step 4: Create a Protected Route

Add a Route for Protected Resources

```
const express = require('express');
const authenticateToken =
require('../middleware/authenticateToken');
const router = express.Router();

router.get('/protected', authenticateToken, (req, res) => {
    res.json({ message: `Welcome, ${req.user.username}` });
});

module.exports = router;
```

Step 5: Initialize the Server

server.js

```
require('dotenv').config();
const express = require('express');
const bodyParser = require('body-parser');
const { sequelize } = require('./config/database');
const authRoutes = require('./routes/authRoutes');
const protectedRoutes = require('./routes/protectedRoutes');

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(bodyParser.json());

// Routes
app.use('/auth', authRoutes);
app.use('/api', protectedRoutes);

// Start the server
(async () => {
    try {
        await sequelize.sync();
        console.log('Database synced');
```

```
        app.listen(PORT, () => {
            console.log(`Server running at
http://localhost:${PORT}`);
        });
    } catch (err) {
        console.error('Error starting the server:', err);
    }
})();
```

Step 6: Environment Variables

.env

```
DB_NAME=your_database_name
DB_USER=your_username
DB_PASSWORD=your_password
DB_HOST=localhost
JWT_SECRET=your_jwt_secret
```

API Endpoints

1. **Register User**:
    - **POST** `/auth/register`

Body:

```
{
  "username": "johndoe",
  "password": "securepassword"
}
```

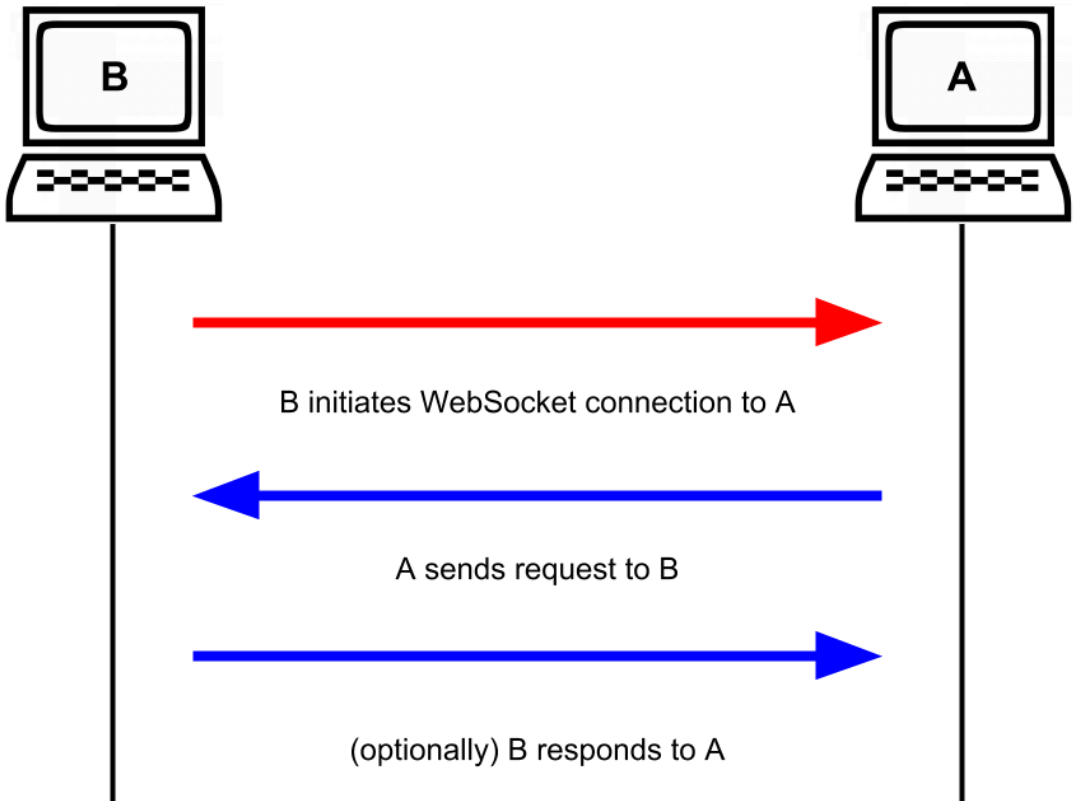
2. **Login User**:
    - **POST** `/auth/login`

Body:

```
{
  "username": "johndoe",
  "password": "securepassword"
}
```

3. **Access Protected Route**:
    - **GET** `/api/protected`
    - Header: `Authorization: Bearer <your_token>`

Chapter 12: Working with WebSockets for Real-Time Communication



WebSockets allow two-way communication between the server and the client, enabling real-time updates and interactions. In this chapter, you will learn how to:

1. Understand WebSockets and their use cases.
2. Set up a WebSocket server in Node.js.
3. Implement real-time communication in a sample chat application.

---

1. What Are WebSockets?

- **WebSocket Protocol**: A communication protocol providing full-duplex communication channels over a single TCP connection.

- **Use Cases**:
    - Real-time chat applications.
    - Live notifications.
    - Online gaming.
    - Collaborative tools like Google Docs.

---

2. Setting Up a WebSocket Server

Node.js provides WebSocket support through libraries like `ws` and `socket.io`.

Install the Required Library

We will use `ws`, a lightweight WebSocket library.

```
npm install ws
```

---

3. Creating a WebSocket Server

Basic WebSocket Server Example

```
const WebSocket = require('ws');

const server = new WebSocket.Server({ port: 8080 });

server.on('connection', (socket) => {
    console.log('Client connected');
```

```
    // Handle incoming messages
    socket.on('message', (message) => {
        console.log('Received:', message);

        // Echo the message back to the client
        socket.send(`Server received: ${message}`);
    });

    // Handle disconnection
    socket.on('close', () => {
        console.log('Client disconnected');
    });
});

console.log('WebSocket server running on ws://localhost:8080');
```

Test the WebSocket Server

You can use a WebSocket client like websocat or browser DevTools to connect:

```
const socket = new WebSocket('ws://localhost:8080');
socket.onmessage = (event) => console.log(event.data);
socket.send('Hello, server!');
```

---

4. Building a Real-Time Chat Application

We'll create a simple chat app using WebSockets.

Directory Structure

```
websocket-chat/
├── public/
│   ├── index.html
│   └── chat.js
├── server.js
```

Step 1: Server-Side WebSocket Implementation

server.js

```
const WebSocket = require('ws');
const server = new WebSocket.Server({ port: 8080 });

let clients = [];

server.on('connection', (socket) => {
    clients.push(socket);
    console.log('New client connected');

    socket.on('message', (message) => {
        console.log('Received:', message);

        // Broadcast the message to all connected clients
        clients.forEach((client) => {
            if (client !== socket && client.readyState ===
```

```
WebSocket.OPEN) {
                client.send(message);
            }
        });
    });

    socket.on('close', () => {
        console.log('Client disconnected');
        clients = clients.filter((client) => client !== socket);
    });
});

console.log('Chat WebSocket server running on
ws://localhost:8080');
```

Step 2: Client-Side Implementation

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>WebSocket Chat</title>
    <link rel="stylesheet" href="styles.css">
```

```
</head>
<body>
    <div id="chat-container">
        <div id="messages"></div>
        <input type="text" id="message-input" placeholder="Type
a message..." />
        <button id="send-button">Send</button>
    </div>
    <script src="chat.js"></script>
</body>
</html>
```

public/chat.js

```
const socket = new WebSocket('ws://localhost:8080');
const messagesDiv = document.getElementById('messages');
const messageInput = document.getElementById('message-input');
const sendButton = document.getElementById('send-button');

// Display incoming messages
socket.onmessage = (event) => {
    const message = document.createElement('div');
    message.textContent = event.data;
    messagesDiv.appendChild(message);
};

// Send messages to the server
```

```
sendButton.addEventListener('click', () => {
    const message = messageInput.value;
    if (message) {
        socket.send(message);
        messageInput.value = '';
    }
});
```

Step 3: Styling the Chat App

public/styles.css

```
body {
    font-family: Arial, sans-serif;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
    background: #f0f0f0;
}

#chat-container {
    width: 300px;
    background: white;
    border: 1px solid #ccc;
    border-radius: 5px;
```

```
    overflow: hidden;
}

#messages {
    height: 300px;
    overflow-y: auto;
    padding: 10px;
    border-bottom: 1px solid #ccc;
}

#message-input {
    width: calc(100% - 50px);
    padding: 10px;
    border: none;
    outline: none;
}

#send-button {
    width: 50px;
    border: none;
    background: #007bff;
    color: white;
    font-weight: bold;
    cursor: pointer;
}

#send-button:hover {
```

```
    background: #0056b3;
}
```

---

5. Running the Application

Start the server:

```
node server.js
```

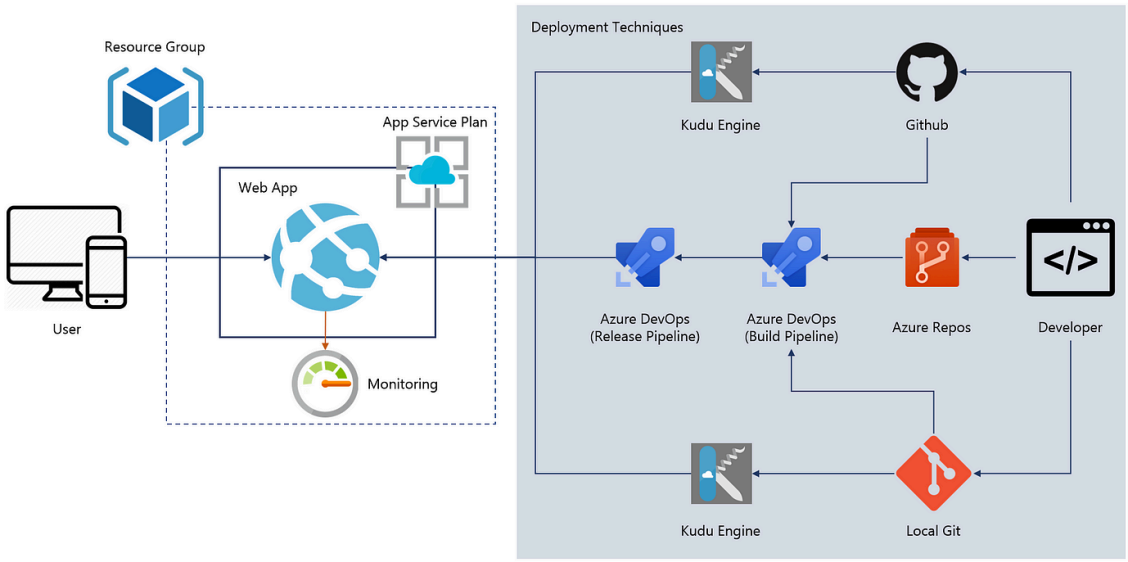
1. Open the `index.html` file in a browser.
2. Open multiple browser windows to test real-time chat functionality.

---

6. Enhancements

1. **Usernames**: Allow users to set and display their names in the chat.
2. **Persistence**: Save chat messages to a database.
3. **Notifications**: Notify users when someone joins or leaves the chat.

Chapter 13: Deploying Node.js Applications



Deploying a Node.js application ensures it runs on a server and is accessible over the internet. In this chapter, you'll learn:

1. Preparing your Node.js app for deployment.
2. Deployment methods and tools.
3. Setting up a production server with **PM2** and **Nginx**.
4. Deploying to platforms like **Heroku**, **Vercel**, or **AWS**.

---

1. Preparing Your App for Deployment

Checklist for Production-Ready Applications:

- **Environment Variables**: Store secrets like API keys and database credentials in `.env` files.
- **Error Handling**: Add robust error handling to your app.
- **Static Assets**: Ensure all public assets (images, CSS, JavaScript) are properly configured.

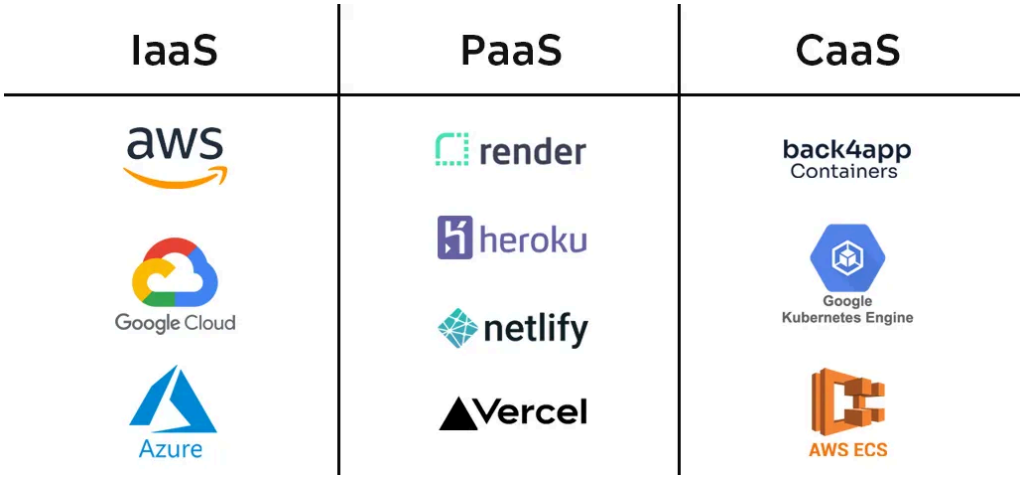
- **Build and Minify**: Minify your JavaScript and CSS files if applicable.

Example `.env` Configuration:

```
PORT=3000
DB_HOST=your_database_host
DB_USER=your_database_user
DB_PASSWORD=your_database_password
JWT_SECRET=your_jwt_secret
```

---

2. Deployment Methods and Tools



Common Deployment Platforms:

- **Vercel**: Ideal for front-end and serverless applications.
- **Heroku**: Simple deployment for Node.js apps with add-ons.
- **AWS** (EC2, Elastic Beanstalk): Scalable solutions for large-scale apps.
- **DigitalOcean**: Offers VPS (virtual private servers).
- **Render**: A developer-friendly cloud platform.

Deployment Tools:

- **Git**: Version control to push changes to remote repositories.
- **Docker**: Containerize your application for consistency.
- **CI/CD Pipelines**: Automate testing and deployment.

---

3. Deploying to Heroku

Step 1: Install Heroku CLI

```
npm install -g heroku
```

Step 2: Login and Initialize

```
heroku login
heroku create your-app-name
```

Step 3: Add a `Procfile`

Create a `Procfile` in your project root:

```
web: node server.js
```

Step 4: Deploy Your Application

```
git init
git add .
git commit -m "Initial commit"
git push heroku main
```

Step 5: View Your App

```
heroku open
```

---

4. Setting Up a VPS with PM2 and Nginx

Step 1: Provision a VPS

Use a provider like **DigitalOcean** or **AWS** to create a VPS.

Step 2: Install Node.js

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash
-
sudo apt install -y nodejs
```

Step 3: Install PM2

PM2 manages your Node.js application processes.

```
npm install -g pm2
pm2 start server.js --name "my-app"
pm2 save
pm2 startup
```

Step 4: Configure Nginx

Install Nginx and set up a reverse proxy.

```
sudo apt install nginx
sudo nano /etc/nginx/sites-available/default
```

Update the configuration:

```
server {
    listen 80;
    server_name your-domain.com;
```

```
    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Restart Nginx:

```
sudo systemctl restart nginx
```

---

5. Deploying to Vercel

Step 1: Install Vercel CLI

```
npm install -g vercel
```

Step 2: Deploy

```
vercel
```

Step 3: Follow Prompts

Answer the prompts to deploy your application.

---

6. Deploying to AWS

Option 1: AWS Elastic Beanstalk

Install the AWS Elastic Beanstalk CLI.

```
pip install awsebcli --upgrade
```

1. Initialize your app.

```
eb init
```

2. Deploy.

```
eb create
```

Option 2: AWS EC2

- Provision an EC2 instance.
- Set up Node.js, PM2, and Nginx.

---

7. Monitoring and Scaling

Use **PM2** for logs and monitoring:

```
pm2 logs
pm2 monit
```

- Scale horizontally with **load balancers** or vertically by upgrading server specs.

What's Next?

Congratulations! You've completed the Node.js course. Here's how you can build on this knowledge:

1. **Explore Advanced Topics**:
    - Microservices architecture.
    - GraphQL APIs.
    - Serverless functions.
2. **Build Projects**:
    - Real-time collaborative tools.
    - E-commerce platforms.
    - Scalable REST APIs.
3. **Certifications**: Get certified in Node.js or backend development.

Chapter 14: Final Guidance for Node.js Mastery

Now that you've completed the Node.js journey, here's a comprehensive guide to best practices, resources, and next steps to solidify your knowledge and take your skills to the next level.

---

1. Best Practices in Node.js Development

Code Organization

- **Modularization**: Use modules to organize code into smaller, reusable chunks.

Directory Structure: Follow a clean directory structure:

```
/src
  /controllers
  /models
  /routes
  /services
  /middlewares
/config
/public
/tests
server.js
```

Error Handling

- Use `try-catch` blocks for synchronous code and `.catch()` for promises.
- Implement a centralized error-handling middleware for Express apps.
- Log errors using tools like **Winston** or **Pino**.

Security

- Validate user inputs to prevent injection attacks.

Use `helmet` for HTTP header security:

```
npm install helmet
```

```
const helmet = require('helmet');
app.use(helmet());
```

-
- Avoid exposing sensitive data; use `.env` for secrets and configuration.
- Use **bcrypt** for password hashing and never store plaintext passwords.

Performance

- Use caching for frequently requested data (e.g., Redis).
- Avoid blocking the event loop; use asynchronous functions where applicable.

Enable Gzip compression for responses using `compression` middleware.

```
const compression = require('compression');
app.use(compression());
```

Database Management

- Use an ORM/ODM like **Sequelize** (SQL) or **Mongoose** (MongoDB).
- Write efficient database queries and use indexes where appropriate.

Testing

- Write unit tests for individual modules and integration tests for APIs.

Use libraries like **Jest**, **Mocha**, and **Supertest**.

```
npm install jest supertest
```

Version Control

- Commit regularly with meaningful messages.
- Use branching strategies like GitFlow for team collaboration.

---

2. Must-Have Tools and Libraries

Utilities

- **Lodash**: Utility functions for manipulating arrays, objects, and more.
- **Moment.js/Day.js**: Handle dates and times effectively.

Authentication

- **Passport.js**: Authentication middleware for Node.js.
- **JWT (jsonwebtoken)**: Token-based authentication.

Validation

- **Joi**: Schema validation for input data.
- **Express Validator**: Middleware for validating request data in Express.

Monitoring and Debugging

- **PM2**: Process manager with monitoring capabilities.
- **Node Inspector**: Debugging tool integrated with Chrome DevTools.

---

3. Recommended Projects

Building projects is the best way to consolidate your learning. Here are some suggestions:

1. **Task Management App**: Use a database for CRUD operations and authentication.
2. **E-commerce Platform**: Implement product listing, shopping cart, and payment integration.
3. **Real-Time Chat App**: Use WebSockets for real-time messaging.
4. **Weather Dashboard**: Fetch data from a weather API and display it.
5. **Blogging Platform**: Create an app with user authentication, posts, and comments.

---

4. Resources for Further Learning

Books

- **"Node.js Design Patterns" by Mario Casciaro and Luciano Mammino**: A deep dive into advanced design patterns.
- **"Eloquent JavaScript" by Marijn Haverbeke**: A solid JavaScript foundation.

Online Courses

- **The Complete Node.js Developer Course (Emmersive Learning)**
- **Node.js and Express.js (Emmersive Learnig)**.

Official Documentation

- Node.js Docs: The go-to resource for all things Node.js.
- Express.js Docs.

Community

- **GitHub**: Explore open-source Node.js projects.
- **Reddit**: Join the r/node community for discussions.
- **Stack Overflow**: Search for solutions to common issues.

---

5. Next Steps and Advanced Topics

Advanced Topics

1. **Microservices**:

    - Split applications into small, independently deployable services.
    - Use message queues like RabbitMQ or Kafka for inter-service communication.
2. **GraphQL**:
    - Build APIs with flexible and efficient data querying.
    - Use libraries like `apollo-server`.
3. **Serverless**:
    - Deploy serverless functions using AWS Lambda, Azure Functions, or Google Cloud Functions.
4. **Event-Driven Architecture**:
    - Use events for decoupled communication between different parts of your app.
5. **Streams**:
    - Work with Node.js streams for handling large files and data efficiently.

Certifications

- Certified Node.js Developer (OpenJS): Validate your skills with a recognized certification.

---

6. Career and Productivity Tips

1. **Build a Portfolio**:
    - Showcase your Node.js projects on a professional portfolio.
    - Include links to GitHub repositories.
2. **Contribute to Open Source**:
    - Find Node.js projects on GitHub and start contributing.
    - It improves your skills and visibility in the community.

3. **Stay Updated**:
    - Follow Node.js changelogs for updates.
    - Regularly read tech blogs like Node.js Medium Blog.
4. **Develop Soft Skills**:
    - Communication and problem-solving skills are critical for teamwork.
    - Practice writing clear and maintainable code.

Final Words of Encouragement

Becoming proficient in Node.js is a journey. Keep practicing, build diverse projects, and never hesitate to ask for help. The Node.js ecosystem is vast and rapidly evolving—continuous learning will keep you ahead in your development career.

Thank you!

Muhammed Teshome