Node Working

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

console.log("Start");

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

console.log("End");
```

Explanation

What Happens Line by Line:

1. **`console.log("Start");`**
   - The first line is a simple synchronous operation.
   - Output: **Start**.
2. **`fs.readFile(...)`**
   - This is an asynchronous operation.
   - Node.js delegates this task to the **File System (fs)** module.
   - The `fs` module sends the file reading operation to the system's I/O layer, which handles it in the background.
   - The callback function `(err, data) => { ... }` is registered to be called **after the file is read**.
3. **`console.log("End");`**
   - Since `fs.readFile` is non-blocking, Node.js does **not wait** for the file to be read.
   - Instead, it immediately executes this line.
   - Output: **End**.
4. **Callback Execution**
   - Once the file reading is completed, the callback `(err, data)` is pushed to the **Callback Queue**.
   - The **Event Loop** ensures that this callback is executed **after all synchronous code has completed**.
   - The file content is then logged to the console: **File Content: [content of example.txt]**.

Output

If the file `example.txt` contains the text `"Hello, Node.js!"`, the output will be:

```
Start
End
```

```
File Content: Hello, Node.js!
```

Execution Flow Diagram

1. **Synchronous Task 1**: Print "Start".
    - **Output**: `Start`.
2. **Asynchronous Task**: Read `example.txt`.
    - Node.js delegates this task to the system's I/O and moves on without waiting.
3. **Synchronous Task 2**: Print "End".
    - **Output**: `End`.
4. **Callback Queue**: Once the file is read, the callback `(err, data)` is queued.
5. **Event Loop**: The callback is executed after all synchronous tasks are completed.
    - **Output**: `File Content: Hello, Node.js!`

Key Points

1. **Non-Blocking Behavior**:
    - The `fs.readFile` operation does not block the execution of the remaining code.
    - This allows the server to remain responsive and handle other tasks simultaneously.
2. **Event Loop and Callback Queue**:
    - The Event Loop ensures the callback runs only after the main thread is idle.
3. **Asynchronous Efficiency**:
    - This approach is ideal for handling multiple I/O-bound tasks, such as reading files, making API calls, or querying databases.

Imagine a single waiter in a restaurant:

- The waiter takes your order (listens for requests).
- If the order is complex (e.g., a dish that takes time to prepare), the waiter sends it to the kitchen (background thread pool).
- Meanwhile, the waiter serves other customers (handles other requests).
- Once your dish is ready, the kitchen informs the waiter, who serves it to you (callback function or promise).

In a technical context, handling millions of requests with a single "waiter" (server) can be challenging due to several factors like limited resources (CPU, memory, network bandwidth) and the need for efficient handling of those requests. However, there are strategies that can help optimize performance:

1. **Concurrency and Asynchronous Processing**: In web servers, many requests are processed concurrently (or asynchronously), meaning the server doesn't need to wait for one request to finish before handling the next. This allows the server to handle multiple requests in parallel, like a waiter taking new orders while serving others.
2. **Load Balancing**: Rather than relying on a single server, the load can be distributed across multiple servers using load balancing techniques. This is like having multiple

waiters handling different sections of the restaurant. Each server (or waiter) handles a subset of requests, improving overall performance and scalability.

3. **Caching**: To reduce the number of requests that need to be processed, frequently requested information can be cached. This is like a waiter memorizing common orders to deliver faster instead of going to the kitchen each time.
4. **Optimized Data Handling**: Efficient algorithms and data structures are crucial for handling large numbers of requests. Servers can optimize how they manage connections, process data, and send responses to ensure minimal delay.
5. **Rate Limiting and Queuing**: Sometimes, servers might limit the number of requests that can be processed at once or queue incoming requests to ensure no one is overloaded. This can prevent a situation where the system becomes overwhelmed.
6. **Distributed Systems**: In large-scale systems, requests might be spread across multiple data centers or services, each handling a part of the request. This is similar to having multiple branches of the restaurant where orders are taken by different waiters and sent to various kitchens.

Buffering vs Chunking:

- **Buffering**: In traditional applications, when you deal with large files or data, the entire file is read or written at once. This is called buffering, and it consumes more memory because you need to store the entire data in memory before it can be processed or sent.
- **Chunking**: In Node.js, chunking breaks down the data into smaller pieces and processes them as they are available. It doesn't require the entire file to be in memory at once, which is why it's more memory-efficient for large data sets.

When Node.js is released under the **MIT License**, it means that the software is open-source and anyone can freely use, modify, distribute, and even sell the software, with very few restrictions. The **MIT License** is one of the most permissive open-source licenses available.

Here's what it means in practical terms:

Key Points of the MIT License:

1. **Free Use**: You can use Node.js for any purpose (commercial, personal, educational, etc.) without paying for a license or royalties.
2. **Modification**: You can modify the Node.js source code to suit your needs. If you want to add features or fix bugs, you're free to do so.
3. **Distribution**: You can distribute the software as-is or with your modifications, either commercially or non-commercially, without any restrictions.
4. **Private or Public Usage**: You can use Node.js both in private projects or as part of a public service, such as in production applications or software for sale.
5. **No Warranty**: The license disclaims any warranties. This means that if something goes wrong when using or modifying Node.js, the authors are not responsible for any issues or damages that arise from using the software.

---

1. Global Installation

- **Definition**: When a package is installed globally, it becomes available system-wide and can be used in any project or directly from the command line.
- **When to Use**: You generally install tools like linters, CLI tools, or utilities globally (e.g., `npm`, `nodemon`, `typescript`).

Commands:

- **Install a Package Globally**:

  ```
  npm install -g <package_name>
  ```

  Example:

  ```
  npm install -g nodemon
  ```

- **Check Installed Global Packages**:

  ```
  npm list -g --depth=0
  ```

- **Remove a Global Package**:

  ```
  npm uninstall -g <package_name>
  ```

  Example:

  ```
  npm uninstall -g nodemon
  ```

---

2. Uninstalling a Package

- To remove a package that is no longer required:

Commands:

- **Local Uninstall** (Project-specific):

  ```
  npm uninstall <package_name>
  ```

  Example:

  ```
  npm uninstall lodash
  ```

- **Uninstall and Remove from `package.json`**: If the package is listed in `dependencies` or `devDependencies`, the above command also removes it from `package.json`.

---

3. Updating a Package

To ensure your project uses the latest compatible or specific version of a package:

Commands:

- **Update a Package Locally**:

  ```
  npm update <package_name>
  ```

  Example:

  ```
  npm update axios
  ```

- **Update All Packages in the Project**:

  ```
  npm update
  ```

- **Update a Global Package**:

  ```
  npm update -g <package_name>
  ```

  Example:

  ```
  npm update -g npm
  ```

- **Check for Updates**:

  ```
  npm outdated
  ```

  This lists outdated packages and their current, wanted, and latest versions.

---

4. Working with Specific Versions of a Package

You might need to install a specific version of a package for compatibility reasons.

Commands:

- **Install a Specific Version**:

  ```
  npm install <package_name>@<version>
  ```

  Example:

  ```
  npm install express@4.17.1
  ```

- **Install the Latest Version**:

  ```
  npm install <package_name>@latest
  ```

- **Install a Version Range**:

  ```
  npm install <package_name>@^1.0.0
  ```

- ^: Accepts minor updates (e.g., `1.x.x`).
- ~: Accepts patch updates (e.g., `1.0.x`).

---

1. npx

- **What It Does**:
  - `npx` is a Node.js package runner that comes with npm (Node Package Manager) 5.2+.
  - It allows you to execute binaries from npm packages without globally installing them on your machine.
- **Use Case**:
  - Run a tool or script from a package without permanently installing it.
- **Example**:

  ```
  npx create-react-app my-app
  ```

  This runs the `create-react-app` package to scaffold a new React project without globally installing the package.

- **Why It's Useful**:
  - Avoids global pollution of packages.
  - Ensures you're using the latest version of the tool when you execute it.

What happens when you use `npx`?

1. **Checks for Local Installation**:
   - `npx` first checks if the package or command is already installed **locally** in your project's `node_modules`.
   - If found, it uses that version.
2. **Checks for Global Installation**:
   - If the package is not found locally, `npx` checks for a **globally installed** version.
   - If a global version exists, it runs that.
3. **Temporary Download**:
   - If the package is not installed locally or globally, `npx` downloads it **temporarily** to a cache directory (usually under `~/.npm/_npx`).
   - After execution, the downloaded package may be deleted or retained in the cache for future use, depending on the package and your environment.

---

2. npm init -f

- **What It Does**:
  - This initializes a new `package.json` file with default values **(force mode)**.
- **Use Case**:
  - Quickly create a `package.json` file without manually entering details.

- **Example**:

  ```
  npm init -f
  ```

  Generates a `package.json` file with default fields.

- **Default Output**:

  ```
  {
    "name": "project-name",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC"
  }
  ```

---

3. npm doctor

- **What It Does**:
  - Diagnoses and checks your Node.js and npm environment for potential issues.
- **Use Case**:
  - Troubleshoot issues with npm or your Node.js setup.
- **Example**:

  ```
  npm doctor
  ```

- **Typical Checks**:
  - Validates the Node.js version.
  - Checks permissions for global directories.
  - Ensures the integrity of npm's configuration and cache.

---

4. npm audit fix

- **What It Does**:
  - Checks your project's dependencies for security vulnerabilities and attempts to fix them automatically.
- **Use Case**:
  - Keep your project secure by updating vulnerable dependencies.
- **Example**:

  ```
  npm audit fix
  ```

- **Important Notes**:
  - Updates only the dependencies listed in `package.json` to resolve vulnerabilities.

- Use `npm audit fix --force` to apply updates that might introduce breaking changes.

---

5. npm ci

- **What It Does**:
  - `npm cache clean`: Clears the npm cache.
  - `npm ci`: Installs dependencies strictly according to the `package-lock.json`.

Bonus: `npm cache clean`

- **What It Does**:
  - Cleans the npm cache to resolve cache corruption or free up space.
- **Use Case**:
  - Use this when you encounter cache-related issues.
- **Example**:

```
npm cache clean --force
```

---

5. `package.json` File

- **Definition**: This is the metadata file for a Node.js project. It contains information about the project, including its dependencies, scripts, and configuration.
- **Purpose**:
  - Manages project dependencies (`dependencies`, `devDependencies`).
  - Specifies project version, name, author, license, etc.
  - Defines custom scripts for tasks (e.g., build, test).

Key Sections:

- **Basic Structure**:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A sample project",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo 'Error: no test specified'"
  },
  "dependencies": {
```

```
    "express": "^4.17.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.15"
  }
}
```

- **Important Fields**:
  - **`dependencies`**: Packages required for the application to run. Example:

    ```
    "dependencies": {
      "express": "^4.17.1"
    }
    ```

  - **`devDependencies`**: Packages only needed for development. Example:

    ```
    "devDependencies": {
      "jest": "^27.0.0"
    }
    ```

  - **`scripts`**: Custom commands to automate tasks. Example:

    ```
    "scripts": {
      "start": "node server.js",
      "test": "jest"
    }
    ```

- **Adding Dependencies**:

  ```
  npm install <package_name> --save
  ```

  This adds the package to `dependencies`.

- **Adding Dev Dependencies**:

  ```
  npm install <package_name> --save-dev
  ```

  This adds the package to `devDependencies`.

---

6. Working with **`package-lock.json`**

- **Purpose**: Ensures consistent installs by locking the exact versions of dependencies.
- **Don't Modify**: It's auto-generated by `npm` and reflects the exact dependency tree.

Core Modules in Node.js

Node.js provides a rich set of core modules that are bundled with the runtime environment. These modules allow developers to perform various operations without installing external packages. Below are some of the most commonly used core modules:

1. Path Module

The `path` module provides utilities for working with file and directory paths. It is particularly useful for building file paths in a platform-independent manner.

Common Methods:

- `path.basename(path)`: Returns the last portion of a path.
- `path.dirname(path)`: Returns the directory name of a path.
- `path.extname(path)`: Returns the extension of the file in a path.
- `path.join([...paths])`: Joins multiple path segments into one.
- `path.resolve([...paths])`: Resolves a sequence of paths into an absolute path.

Example:

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

const filePath = '/Users/john/Documents/file.txt';
console.log(path.basename(filePath)); // Output: file.txt
console.log(path.dirname(filePath)); // Output: /Users/john/Documents
console.log(path.extname(filePath)); // Output: .txt
console.log(path.join('/folder1', 'folder2', 'file.txt')); // Output:
/folder1/folder2/file.txt
```

---

2. Child Process Module

The `child_process` module allows you to spawn new processes to execute commands or scripts. It provides four methods:

Common Methods:

- `exec()`: Executes a command in a shell and buffers the output.
- `spawn()`: Launches a new process with a given command.
- `execFile()`: Similar to `exec()`, but directly executes a file.
- `fork()`: A specialized version of `spawn()` to create Node.js child processes.

Example:

```
const { exec } = require('child_process');

exec('ls', (error, stdout, stderr) => {
    if (error) {
        console.error(`Error: ${error.message}`);
        return;
    }
    if (stderr) {
        console.error(`Stderr: ${stderr}`);
        return;
    }
    console.log(`Output: ${stdout}`);
});
```

---

3. OS Module

The `os` module provides operating system-related utility methods and properties.

Common Methods:

- `os.arch()`: Returns the operating system CPU architecture.
- `os.platform()`: Returns the platform of the operating system.
- `os.cpus()`: Returns information about each CPU/core installed.
- `os.totalmem()`: Returns the total amount of system memory.
- `os.freemem()`: Returns the amount of free system memory.

Example:

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

console.log('Platform:', os.platform());
console.log('Architecture:', os.arch());
console.log('Total Memory:', os.totalmem());
console.log('Free Memory:', os.freemem());
```

---

4. URL Module

The `url` module provides utilities for URL resolution and parsing.

Common Methods:

- `new URL(input[, base])`: Creates a new URL object.
- `url.parse(urlStr, [parseQueryString], [slashesDenoteHost])`: Parses a URL string.

Example:

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

const myURL = new
URL('https://example.com:8080/path/name?query=test#hash');
console.log(myURL.hostname); // Output: example.com
console.log(myURL.pathname); // Output: /path/name
console.log(myURL.search); // Output: ?query=test
console.log(myURL.hash); // Output: #hash
```

---

5. Util Module

The `util` module provides various utility functions.

Common Methods:

- `util.format()`: Formats strings like `printf` in C.
- `util.promisify()`: Converts callback-based functions to return a Promise.

Example:

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

function callbackStyleFunction(a, b, callback) {
    setTimeout(() => {
        callback(null, a + b);
    }, 1000);
}

const promiseFunction = util.promisify(callbackStyleFunction);

promiseFunction(3, 4).then(result => console.log(result)); // Output: 7
```

---

6. Creating a Web Server

The `http` module in Node.js is a core module that allows developers to create and manage HTTP servers. This module is essential for building web servers that handle client requests and send responses.

Key Concepts of the HTTP Module

1. **HTTP Requests and Responses**:
    - **Request**: Data sent by the client to the server (e.g., GET, POST, PUT, DELETE).
    - **Response**: Data sent by the server back to the client.
2. **Listening on a Port**:
    - Servers listen on a specific port to handle incoming requests.
    - Common ports: 80 (HTTP), 443 (HTTPS), 3000 (development).
3. **Basic Structure**:
    - `http.createServer(callback)`: Creates an HTTP server that listens for requests.
    - The callback receives two objects:
        - `req` (request): Represents the incoming request.
        - `res` (response): Represents the server's response.

Creating a Simple HTTP Server

Here's a simple example:

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

// Create an HTTP server
const server = http.createServer((req, res) => {
    // Set the response header
    res.writeHead(200, { 'Content-Type': 'text/plain' });

    // Send the response body
    res.end('Hello, World!\n');
});
```

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

Explanation:

1. **`http.createServer`**:
    - Sets up the server and listens for client requests.
2. **Response Headers**:
    - `res.writeHead(200, { 'Content-Type': 'text/plain' })`: Sends a status code (`200 OK`) and content type.
3. **Response Body**:
    - `res.end('Hello, World!\n')`: Sends the response and ends the connection.
4. **Server Listening**:
    - `server.listen(3000)`: The server listens on port 3000.

Advanced Features of HTTP Server

Routing

Routing allows you to respond differently based on the request URL.

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

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

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

Serving Static Files

To serve files like HTML, CSS, or JavaScript:

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

const server = http.createServer((req, res) => {
    if (req.url === '/') {
        fs.readFile('index.html', (err, data) => {
            if (err) {
```

```
                res.writeHead(500);
                res.end('Error loading the file.');
                return;
            }
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end(data);
        });
    } else {
        res.writeHead(404);
        res.end('Page not found.');
    }
});

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

---

7. File System Module

The `fs` module provides an API for interacting with the file system.

Common Methods:

- `fs.readFile(path, callback)`: Reads the content of a file.
- `fs.writeFile(path, data, callback)`: Writes data to a file.
- `fs.appendFile(path, data, callback)`: Appends data to a file.
- `fs.unlink(path, callback)`: Deletes a file.

Example:

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

// Writing to a file
fs.writeFile('example.txt', 'Hello, Node.js!', (err) => {
    if (err) throw err;
    console.log('File written!');

    // Reading the file
    fs.readFile('example.txt', 'utf8', (err, data) => {
        if (err) throw err;
        console.log('File content:', data);
    });
});
```

These core modules form the foundation of Node.js, enabling developers to build powerful and efficient server-side applications. Understanding and leveraging them effectively is key to mastering Node.js.

Using the `events` Module

Importing the Module

To use events, you need to import the `events` module and create an instance of `EventEmitter`:

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

Basic Example

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

// Create an instance of EventEmitter
const myEmitter = new EventEmitter();

// Add an event listener for 'greet' event
myEmitter.on('greet', (name) => {
    console.log(`Hello, ${name}!`);
});

// Emit the 'greet' event
myEmitter.emit('greet', 'India');
```

Output:

```
Hello, India!
```

Key Methods in `EventEmitter`

1. **`on(event, listener)`**:
   - Registers a listener for an event.
2. **`emit(event, [args])`**:
   - Emits an event and optionally passes arguments to listeners.
3. **`once(event, listener)`**:
   - Registers a one-time listener for an event, which is removed after being triggered.
4. **`removeListener(event, listener)`**:
   - Removes a specific listener from an event.
5. **`removeAllListeners([event])`**:
   - Removes all listeners for a specific event (or all events if no event is specified).

Advanced Example: Multiple Events

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

const myEmitter = new EventEmitter();

// Event for order confirmation
myEmitter.on('order', (orderId) => {
    console.log(`Order ${orderId} has been confirmed.`);
});
```

```
// Event for shipping update
myEmitter.on('shipping', (orderId, status) => {
    console.log(`Order ${orderId} is now ${status}.`);
});

// Emit the events
myEmitter.emit('order', 101);
myEmitter.emit('shipping', 101, 'shipped');
```

Output:

```
Order 101 has been confirmed.
Order 101 is now shipped.
```

Using `once` for One-Time Events

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

const myEmitter = new EventEmitter();

// Listener will run only once
myEmitter.once('disconnect', () => {
    console.log('User disconnected.');
});

myEmitter.emit('disconnect'); // Triggers the listener
myEmitter.emit('disconnect'); // Does nothing
```

Output:

```
User disconnected.
```

Error Handling in Events

If an event listener throws an error, it can crash the application unless handled. Node.js provides a special `error` event for this purpose.

Example: Handling Errors

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

const myEmitter = new EventEmitter();

// Error handling
myEmitter.on('error', (err) => {
    console.error(`Error occurred: ${err.message}`);
});

// Emit an error event
myEmitter.emit('error', new Error('Something went wrong!'));
```

Output:

```
Error occurred: Something went wrong!
```

Real-World Use Cases

1. **HTTP Server Example**:
    - Events like `request`, `connection`, and `close` are commonly used.

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

// Listener for 'request' event
server.on('request', (req, res) => {
    console.log(`Request received: ${req.url}`);
    res.end('Hello, Node.js!');
});

// Start the server
server.listen(3000, () => {
    console.log('Server listening on port 3000.');
});
```

---

2. **Custom Application Logic**:
    - Events are used to decouple different parts of an application.
    - For example, notifying when data is received or processed.

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

const app = new EventEmitter();

app.on('data_received', (data) => {
    console.log('Data received:', data);
    app.emit('process_data', data);
});

app.on('process_data', (data) => {
    console.log('Processing data:', data.toUpperCase());
});

// Simulate receiving data
app.emit('data_received', 'hello');
```

Output:

```
Data received: hello
Processing data: HELLO
```