Node.js is a **JavaScript runtime environment** built on **Chrome's V8 JavaScript engine**. It allows developers to execute JavaScript code outside the browser, making it a popular choice for building **server-side applications**.

Key Features of Node.js:

Asynchronous and Event-Driven

- Node.js uses a non-blocking I/O model, making it efficient for handling multiple requests concurrently.

Single-Threaded

- It operates on a single thread but can handle multiple operations asynchronously using the event loop.

Fast and Scalable

- Powered by the V8 engine, Node.js can execute JavaScript code at high speed, making it suitable for scalable applications.

Rich Ecosystem with NPM

- The **Node Package Manager (NPM)** provides a vast library of reusable modules that accelerate development.

Cross-Platform

- Node.js works on Windows, macOS, and Linux, making it versatile for different environments.

QUES: Node.js uses a non-blocking I/O model, making it efficient for handling multiple requests concurrently. Explain.

Ans - The non-blocking I/O model in Node.js means that it can handle multiple input/output operations simultaneously without waiting for one operation to complete before starting the next. This approach makes Node.js highly efficient, especially for I/O-intensive applications like web servers or APIs.

How Non-Blocking I/O Works:

1. **Traditional Blocking I/O (Synchronous):**
    - In traditional models, when a program performs an I/O operation (e.g., reading a file or fetching data from a database), the entire program pauses until that operation completes. This blocks the program from handling other tasks during the wait time.

Example:

```javascript
const fs = require('fs');
const data = fs.readFileSync('file.txt', 'utf-8'); // Blocking
console.log(data);
console.log('File read complete.');
```

Output:

- The file's content is logged first.
- The next line (`File read complete`) executes only after the file is completely read.

Non-Blocking I/O in Node.js:

- In Node.js, when an I/O operation is initiated, it doesn't block the main thread. Instead, the operation is offloaded (e.g., to the OS or a worker thread), and the program continues executing other tasks.
- Once the I/O operation completes, a callback function is triggered to handle the result.

The Event Loop:

Node.js uses an **event loop** to manage non-blocking operations. The event loop:

1. Receives I/O tasks or asynchronous callbacks.
2. Offloads these tasks to the system or worker threads.
3. Keeps processing other tasks in the queue.
4. Executes the callback once the I/O task finishes.

Benefits:

- **Efficiency:** The program can handle many requests at the same time because it doesn’t wait for I/O operations to complete.
- **Scalability:** Ideal for high-traffic applications like APIs, where multiple users send requests simultaneously.
- **Responsiveness:** Faster responses to users since no thread is blocked.

Here's a larger example of a **non-blocking operation** where a Node.js server handles multiple HTTP requests simultaneously.

javascript

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

const server = http.createServer((req, res) => {
  if (req.url === '/') {
    // Non-blocking file read
    fs.readFile('example.txt', 'utf-8', (err, data) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end('Server Error');
      } else {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end(data);
      }
    });
    console.log('File reading initiated for / request');
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

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

How This Works:

1. **Server Starts:**
   - The server listens on port `3000` and waits for incoming requests.
2. **Handling Requests:**
   - When a user accesses `/`, the server initiates reading `example.txt` using a **non-blocking** file operation.
   - While the file is being read, the server **continues handling other requests** (if any).
3. **Callback Execution:**
   - Once the file is read, its content is sent as the response.
4. **404 Handling:**
   - Any request other than `/` receives a 404 response

Output:

- User accesses `http://localhost:3000/`:
  - The server logs

```
File reading initiated for / request
```

- The content of `example.txt` is sent to the browser.

User accesses an unknown route, e.g., `http://localhost:3000/unknown`:

- The server responds with `404 Not Found`.

Key Point:

This example demonstrates how Node.js handles **file I/O and multiple HTTP requests simultaneously** without blocking the main thread. Even while reading a file, the server is ready to respond to other requests.

Let's break the code into smaller sections and explain how each part executes step-by-step:

1. **Importing Required Modules**

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

`http` module: Allows us to create a web server.
`fs` module: Provides file system operations like reading and writing files.

2. Creating the Server

```
javascript

const server = http.createServer((req, res) => {
```

`http.createServer`: **Creates a server that listens for incoming HTTP requests.**
The callback `(req, res)`:

- `req`: **Contains information about the incoming request (e.g., URL).**
- `res`: **Used to send a response back to the client.**

3. Handling Requests

```
if (req.url === '/') {
```

Checks if the user is accessing the root URL (`/`). If so, execute the code inside this block.

4. Reading a File Asynchronously

```
fs.readFile('example.txt', 'utf-8', (err, data) => {
```

`fs.readFile`: **Starts reading the file** `example.txt`.
Arguments:

- `'example.txt'`: **File to be read.**
- `'utf-8'`: **Encoding to read the file as text.**
- **Callback** `(err, data)`: **Executes once the file is read.**
  - `err`: **Contains an error if the read operation fails.**
  - `data`: **Contains the file's content if the operation is successful.**

5. Handling File Read Results

```
if (err) {
  res.writeHead(500, { 'Content-Type': 'text/plain' });
  res.end('Server Error');
} else {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end(data);
}
```

Inside the callback:

- **If `err` is not `null`, it means the file couldn't be read:**
  - **Respond with a `500` status (Internal Server Error).**
- **If the file is read successfully:**
  - **Respond with a `200` status (OK) and send the file's content (`data`).**

6. Log to Console

```
console.log('File reading initiated for / request');
```

7. Handling Non-Root URLs

```
} else {
  res.writeHead(404, { 'Content-Type': 'text/plain' });
  res.end('Not Found');
}
```

If the request URL is not `/`:

- **Respond with a `404` status (Not Found).**
- **Send the message `"Not Found"` to the client.**

8. Start the Server

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

`server.listen(3000)`: Starts the server on port `3000`.
The callback:

- **Logs the message `"Server is running on http://localhost:3000"` when the server starts successfully.**

Execution Flow:

1. **Server Starts:**
   - **The server begins listening on port `3000`.**
   - **The message `"Server is running on http://localhost:3000"` is logged.**
2. **Request Comes In:**
   - **A user sends an HTTP request to `http://localhost:3000`.**
3. **Check URL:**
   - **If the URL is `/`:**
     - **Start reading `example.txt` asynchronously.**
     - **Immediately log `"File reading initiated for / request"`.**
   - **If the URL is not `/`:**
     - **Respond with `404` (Not Found).**
4. **File Reading Completes:**
   - **The `fs.readFile` callback is triggered:**
     - **If there's an error, respond with `500` (Server Error).**
     - **If successful, respond with `200` and the file content.**
5. **Continue Processing:**
   - **While the file is being read, the server can handle other requests because the file read operation is non-blocking.**