Events and Event Loop in Node.js

Node.js uses an **event-driven** architecture, meaning it reacts to events (like user input, file reading, or database access) rather than following a fixed sequence of instructions. The **event loop** is what makes this possible by managing asynchronous operations efficiently.

---

1. Events in Node.js

Events in Node.js work like real-world events. For example, a button click is an event, and you can assign a function to execute when that event happens.

Node.js has an **EventEmitter** class that helps manage events.

Example: Creating and Emitting an Event

```
const EventEmitter = require('events');  // Import events module
const eventEmitter = new EventEmitter(); // Create event object
```

// Define an event and assign a function

```
eventEmitter.on('greet', function() {
  console.log('Hello! Event triggered.');
});
```

// Emit the event

```
eventEmitter.emit('greet');  // Output: Hello! Event triggered.
```

- on('event', callback): Listens for the event.
- emit('event'): Triggers the event.

---

2. Event Loop in Node.js

The **Event Loop** is the core of Node.js's asynchronous behavior. It continuously checks for completed operations and executes their callbacks.

When a task (like file reading) is started, it is sent to the **event queue**, and Node.js moves to the next task. Once the task is done, its callback function is placed in the **event loop**, which executes it.

Example: Event Loop in Action

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

setTimeout(() => {
  console.log("Inside setTimeout");
}, 0);


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

Output:

Start

End

Inside setTimeout

Even though setTimeout() has 0 milliseconds, Node.js executes it **after** other synchronous tasks (console.log("Start") and console.log("End")). This is because the **event loop** handles asynchronous functions last.

Timers in Node.js

Timers in Node.js are used to execute code **after a delay** or **at regular intervals**. Node.js provides built-in **timer functions** that help manage asynchronous execution.

1. setTimeout() – Runs Code After a Delay

The setTimeout() function executes a function **once** after a specified delay (in milliseconds).

Example:

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

setTimeout(() => {
  console.log("Executed after 2 seconds");
}, 2000);

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

Output: Start

End

Executed after 2 seconds

- Even though setTimeout() is set to 2 seconds, Node.js **does not wait** for it.
- It **moves to the next task** (console.log("End")) and executes the timeout function later.

2. setInterval() – Runs Code Repeatedly

The setInterval() function runs a function **repeatedly** at fixed time intervals.

Example:

```
let count = 1;
let interval = setInterval(() => {
  console.log("Repeated execution", count);
  count++;

  if (count > 5) {
    clearInterval(interval); // Stops execution after 5 times
  }
}, 1000);
```

Output:

Repeated execution 1

Repeated execution 2

Repeated execution 3

Repeated execution 4

Repeated execution 5

- The function runs every **1 second**.
- After 5 executions, clearInterval(interval) stops it.

3. setImmediate() – Runs Code Immediately After I/O

The setImmediate() function executes a function **immediately after the current event loop cycle**.

Example:

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

setImmediate(() => {
  console.log("Executed immediately after I/O");
});
```

console.log("End");

Output:

Start

End

Executed immediately after I/O

- Unlike setTimeout(() => {}, 0), setImmediate() ensures the function runs **as soon as possible after I/O operations**.

4. clearTimeout(), clearInterval(), and clearImmediate()

- clearTimeout(timeoutID): Cancels a scheduled setTimeout().
- clearInterval(intervalID): Stops a setInterval().
- clearImmediate(immediateID): Cancels a setImmediate().

Example: Cancelling setTimeout

```
let timeout = setTimeout(() => {
  console.log("This will not run");
}, 3000);


clearTimeout(timeout);
```

Since clearTimeout(timeout) is called, the function **never executes**.

Error Handling in Node.js

Error handling in Node.js is important for **detecting, managing, and preventing application crashes**. Since Node.js is asynchronous, errors must be handled properly to avoid unexpected failures.

Types of Errors in Node.js

1. **Syntax Errors** – Mistakes in code syntax.
2. **Runtime Errors** – Errors occurring while the program runs (e.g., undefined variables).
3. **Logical Errors** – Code runs but gives the wrong result (hardest to detect).
4. **Operational Errors** – Failures due to system-related issues (e.g., file not found, network failure).

1. Try-Catch for Synchronous Errors

For **synchronous** (blocking) code, errors are handled using try...catch.

Example: Handling an Error in Synchronous Code

```
try {
  let x = undefinedVariable;  // This will throw an error
} catch (error) {
  console.error("An error occurred:", error.message);
}
```

Output:

An error occurred: undefinedVariable is not defined

- The try block **runs the code**.
- If an error occurs, the catch block **catches and handles** the error.

---

2. Handling Errors in Asynchronous Code

Since Node.js runs asynchronous operations (like reading a file), errors must be handled differently.

Example: Handling an Error in Asynchronous Code

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

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

Output:

Error reading file: ENOENT: no such file or directory, open 'nonexistent.txt'

- If fs.readFile() fails, the err object contains details about the error.

---

3. Using Promises for Error Handling

If using **Promises**, errors are handled with .catch().

Example: Handling an Error in Promises

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

fs.readFile('nonexistent.txt', 'utf8')
  .then(data => console.log(data))
  .catch(error => console.error("Promise Error:", error.message));
```

- If the file does not exist, .catch() handles the error.

---

4. Using Async/Await with Try-Catch

For modern async functions, use try...catch inside async functions.

Example: Error Handling in Async/Await

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

async function readFile() {
  try {
    let data = await fs.readFile('nonexistent.txt', 'utf8');
    console.log(data);
  } catch (error) {
    console.error("Async Error:", error.message);
  }
}

readFile();
```

- If an error occurs, catch handles it gracefully.

---

5. Handling Uncaught Exceptions

To **prevent the application from crashing**, use process.on('uncaughtException') to handle unexpected errors.

Example: Catching Uncaught Errors

```
process.on('uncaughtException', (err) => {
  console.error("Unhandled Exception:", err.message);
});


throw new Error("Something went wrong!");  // This error will be
caught
```

- **Avoid using this for normal error handling**; it is mainly used for critical issues.

Buffers in Node.js

In Node.js, a **Buffer** is a temporary storage space for raw binary data. Buffers are used to **handle binary data** directly, especially when dealing with streams such as **files, network requests**, or **TCP streams**.

Buffers are useful because JavaScript (by default) doesn't handle raw binary data well, and Node.js provides this support through Buffers.

Why Buffers Are Needed

- Node.js is often used to handle data from files or networks.
- These data sources send information in **chunks**, not all at once.
- Buffers store these chunks in **binary format** until the entire data is received or processed.

Creating Buffers

1. Create a buffer with a fixed size:

```
const buffer = Buffer.alloc(10);  // Creates a buffer of 10 bytes
filled with zeros
console.log(buffer);
```

2. Create buffer from a string:

```
const buffer = Buffer.from("Hello");
console.log(buffer);             // <Buffer 48 65 6c 6c 6f>
console.log(buffer.toString()); // Hello
```

Useful Buffer Methods

1. .length – Size of the buffer in bytes

```
const buffer = Buffer.from("NodeJS");
console.log(buffer.length);  // 6
```

2. .toString() – Converts buffer data to string

```
console.log(buffer.toString()); // NodeJS
```

3. .write() – Writes to the buffer

```
const buffer = Buffer.alloc(10);
buffer.write("Hi");
console.log(buffer.toString()); // Hi
```

Example: Reading a File in Chunks Using Buffers

```
const fs = require('fs');
const readStream = fs.createReadStream('sample.txt');

readStream.on('data', (chunk) => {
  console.log("Received chunk:", chunk.toString());
});
```

- Here, the file is read **chunk by chunk**.
- Each chunk is a **Buffer**, which we convert to a string for display.

Streams in Node.js

Streams are a powerful feature in Node.js used to handle **reading or writing data in chunks**, instead of loading the entire data into memory at once. This is especially useful for **large files, network data, or real-time data processing**.

Why Use Streams?

- Efficient memory usage
- Faster processing of large files
- Suitable for real-time data (e.g., video/audio streaming, file upload/download)

Types of Streams in Node.js

1. **Readable** – Used to **read** data
2. **Writable** – Used to **write** data
3. **Duplex** – Can **read and write** (e.g., TCP sockets)
4. **Transform** – Like Duplex, but can also **modify** data while reading/writing (e.g., compression)

1. Readable Stream Example

Reading a file using stream:

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

const readStream = fs.createReadStream('example.txt', 'utf8');
```

```
readStream.on('data', (chunk) => {
  console.log('Received chunk:', chunk);
});

readStream.on('end', () => {
  console.log('Reading completed.');
});
```

- data event: triggers when a chunk is read
- end event: triggers when reading is finished

---

2. Writable Stream Example

Writing data to a file using stream:

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

const writeStream = fs.createWriteStream('output.txt');

writeStream.write('Hello, ');
writeStream.write('this is a stream example.');
writeStream.end();  // Signals that writing is complete
```

---

3. Piping Streams

Piping allows direct connection between a **readable** and a **writable** stream.

Example: Copying File Content

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

const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');

readStream.pipe(writeStream);
```

- The content from input.txt is directly piped to output.txt.
- This method is efficient and easy to use.

---

4. Transform Stream Example

A transform stream can change the data while streaming. For example, converting data to uppercase.

```
const { Transform } = require('stream');

const upperCaseTransform = new Transform({
  transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
});
```

This can be used in a pipeline to **modify streaming data**.

Working with the File System in Node.js

Node.js provides a built-in module called fs (short for **File System**) which allows you to work with files and directories. Using this module, you can **create, read, write, update, delete, and rename** files easily.

1. Import the fs module

Before using the file system, import it in your code:

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

2. Reading a File

Asynchronous Method (Recommended)

This method reads the file **without blocking** other operations.

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

- 'utf8' ensures you get readable text.
- data contains the file's content.

Synchronous Method

This method **blocks** the program until the file is read completely.

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

3. Writing to a File

This will **create a new file** or **replace content** if the file already exists.

Asynchronous:

```
fs.writeFile('output.txt', 'This is new content', (err) => {
  if (err) {
    console.error("Error writing file:", err);
    return;
  }
  console.log("File written successfully");
});
```

---

4. Appending Data to a File

Use this when you want to **add more content** to an existing file instead of replacing it.

```
fs.appendFile('output.txt', '\nMore data added.', (err) => {
  if (err) throw err;
  console.log('Data appended to file');
});
```

---

5. Deleting a File

Deletes the specified file.

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

---

6. Renaming a File

Changes the name of a file.

```
fs.rename('oldname.txt', 'newname.txt', (err) => {
  if (err) throw err;
  console.log('File renamed successfully');
});
```

---

7. Working with Directories

Create a New Directory

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

Read Contents of a Directory

Lists files and folders in a specified path.

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

Remove a Directory

```
fs.rmdir('newFolder', (err) => {
  if (err) throw err;
  console.log('Directory removed');
});
```
