Node.js

Streams in Node.js

Part 2: Types & Advanced Operations

March 23, 2025

Source Code

NODE.JS | STREAMS

1. Types of Streams

Node.js provides four fundamental types of streams:

- **Readable:** Sources from which data can be read (files, HTTP requests)
- **Writable:** Destinations to which data can be written (files, HTTP responses)
- **Duplex:** Both readable and writable (TCP sockets)
- **Transform:** Modify data as it passes through (compression, encryption)

```
// Basic examples of stream types

// Readable Stream
const fs = require("fs");
const readableStream = fs.createReadStream("file.txt");
readableStream.on("data", (chunk) => {
  console.log(‘Received ${chunk.length} bytes‘);
});

// Writable Stream
const writableStream = fs.createWriteStream("output.txt");
writableStream.write("Hello World\n");
writableStream.end();
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
14
15 // Transform Stream
16 const { Transform } = require("stream");
17 new Transform({
18   transform(chunk, encoding, callback) {
19     // Convert data to uppercase
20     callback(null, chunk.toString().toUpperCase());
21   },
22 });
```

2. Stream Operations

2.1. The Pipe Method

The most powerful way to connect streams:

```
1 const fs = require('fs');
2 const zlib = require('zlib');
3
4 // Creating a pipeline using pipe()
5 fs.createReadStream('file.txt')
6   .pipe(zlib.createGzip())
7   .pipe(fs.createWriteStream('file.txt.gz'))
8   .on('finish', () => {
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
    console.log('Compression completed');
  });
```

2.2. Pipeline: Improved Error Handling

The **pipeline()** function enhances error handling and resource cleanup:

```
const { pipeline } = require('stream');
const fs = require('fs');
const zlib = require('zlib');

// Using pipeline for better error handling
pipeline(
  fs.createReadStream('input.txt'),
  zlib.createGzip(),
  fs.createWriteStream('output.gz'),
  (err) => {
    if (err) {
      console.error('Pipeline failed', err);
    } else {
      console.log('Pipeline succeeded');
    }
  }
);
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

3. Practical Use Cases

3.1. Processing Large Files

Streams excel when working with files that exceed available memory:

```
const fs = require('fs');
const csv = require('csv-parser');

// Process a large CSV file line by line
fs.createReadStream('huge-data.csv')
  .pipe(csv())
  .on('data', (row) => {
    // Process each row without loading the entire file
    console.log(row);
  })
  .on('end', () => {
    console.log('Processing complete');
  });
```

3.2. HTTP Streaming

Efficiently serve large files or video content:

```
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

NODE.JS | **STREAMS**

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

const server = http.createServer((req, res) => {
  if (req.url === '/video' && req.method === 'GET') {
    const videoPath = './video.mp4';
    const stat = fs.statSync(videoPath);

    res.writeHead(200, {
      'Content-Length': stat.size,
      'Content-Type': 'video/mp4'
    });

    // Stream the file directly to the response
    fs.createReadStream(videoPath).pipe(res);
  } else {
    res.writeHead(404);
    res.end('Resource not found');
  }
});

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

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

4. Best Practices

4.1. Managing Backpressure

Prevent memory overflow when reading faster than writing:

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

const readableStream = fs.createReadStream('large-file.dat');
const writableStream = fs.createWriteStream('destination.dat');

readableStream.on('data', (chunk) => {
  // write() returns false when internal buffer is full
  const canWrite = writableStream.write(chunk);

  if (!canWrite) {
    // Pause the readable stream until the writable drains
    readableStream.pause();

    // Resume when the writable can accept more data
    writableStream.once('drain', () => {
      readableStream.resume();
    });
  }
});
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
20
21 readableStream.on('end', () => {
22   writableStream.end();
23 });
```

4.2. Custom Transform Streams

Create specialized processors for your data:

```
1  const { Transform } = require('stream');
2
3  // Stream to filter lines containing a keyword
4  class LineFilter extends Transform {
5    constructor(keyword) {
6      super();
7      this.keyword = keyword;
8      this.incomplete = '';
9    }
10
11   _transform(chunk, encoding, callback) {
12     // Convert chunk to string and combine with previous data
13     const data = this.incomplete + chunk.toString();
14     // Split by lines
15     const lines = data.split('\n');
16     // Save the last line for the next chunk
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
    this.incomplete = lines.pop();

    // Filter and send lines containing the keyword
    for (const line of lines) {
      if (line.includes(this.keyword)) {
        this.push(line + '\n');
      }
    }
    callback();
  }

  _flush(callback) {
    // Process any remaining data
    if (this.incomplete && this.incomplete.includes(this.keyword)) {
      this.push(this.incomplete + '\n');
    }
    callback();
  }
}
```

5. Conclusions

5.1. Powerful Data Processing Pipelines

Node.js streams provide a versatile framework for building efficient data processing pipelines. By connecting different stream types through piping or the pipeline API, developers can create complex data

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

workflows that process information incrementally. This architecture naturally fits many real-world problems, from ETL processes to real-time data transformations. The ability to compose streams together like building blocks makes it possible to create maintainable solutions that can evolve with changing requirements.

5.2. Enhanced Application Robustness

The pipeline API and proper handling of backpressure significantly improve application reliability. These techniques ensure that data flows smoothly between streams without overwhelming memory resources. By implementing error handling at each stage of the stream pipeline, applications can gracefully recover from failures and ensure proper resource cleanup. These practices are essential for building production-grade systems that can handle unexpected conditions and maintain performance under varying loads.

5.3. Domain-Specific Solutions

The ability to create custom Transform streams unlocks Node.js's streaming capabilities for specific application domains. By extending the standard stream classes, developers can implement specialized data processing logic that maintains all the benefits of the streaming architecture. This approach enables the creation of reusable components that can be integrated into larger stream pipelines, promoting code reuse and separation of concerns. Custom streams represent the full potential of Node.js's streaming model when applied to unique business problems.

6. References

- Node.js. (2023). *Stream | Node.js v18.x Documentation*. Link
- NodeSource. (2022). *Understanding Streams in Node.js*. Link
- Alapont, R. (2023). *Streamlining Your Code: Best Practices for Node.js Streams*. Link
- Alapont, R. (2023). *Error Handling in Node.js Streams: Best Practices*. Link

---

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- Clarion Technologies. (2022). *Node.js for Real-Time Data Streaming*. Link
- Translated, Edited and written in collaboration with AI.

7. Explore My Other Posts

Enjoyed This Content?

Don't miss my previous post about:

Node.js Streams: Part 1 - Introduction & Memory Efficiency

Learn the fundamentals of Node.js Streams and discover how they can dramatically reduce memory usage when processing large files.

Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

www.asanchezyali.com

Ready to Transform Your Data Flow?