**📘Streams in Node.js**

In **Node.js**, **streams** are objects that let you **read data from a source** or **write data to a destination** in a continuous fashion. They are a powerful way to handle **I/O operations** (e.g., reading files, network communications, or stdin/stdout), especially when working with **large amounts of data**.

**🔄 Why Use Streams?**

Imagine reading a **large file**. Without streams, Node.js might load the whole file into memory before processing it, which is inefficient and can crash your app for very large files.

Streams solve this by processing data **piece by piece** (in **chunks**) as it's available, reducing memory usage and increasing performance.

**🚀 Types of Streams in Node.js**

There are **4 main types** of streams:

| **Stream Type** | **Description** | **Example Use Case** |
| --- | --- | --- |
| **Readable** | Stream from which data can be read | Reading a file  (e.g., fs.createReadStream) |
| **Writable** | Stream to which data can be written | Writing to a file  (e.g., fs.createWriteStream) |
| **Duplex** | Both readable and writable | A TCP socket  (e.g., net.Socket) |
| **Transform** | A duplex stream that can modify data | Compression, encryption, etc.  (e.g., zlib.createGzip) |

**🛠️ Stream Events**

Streams are **event-based**, and common events include:

* **data** – When a chunk of data is available.
* **end** – When no more data is available.
* **error** – When an error occurs.
* **finish** – When all data is flushed (for writable streams).

**⚙️ Why Use Streams?**

* **Efficient memory usage** – Data is processed in chunks.
* **Non-blocking** – Enhances performance in large data operations.
* **Composable** – You can chain multiple streams for transformation (e.g., read → compress → write).

**🧩 Real-World Applications**

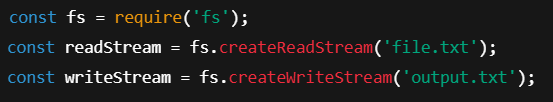
* Streaming large files (e.g., video/audio)
* Handling file uploads/downloads
* Building HTTP servers and proxies
* Real-time data processing

**✅ Conclusion**

Streams are essential in Node.js for building fast, memory-efficient applications, especially when dealing with large or continuous flows of data. By using the right type of stream and combining them smartly, you can dramatically improve the performance of your applications.

**✅ How Streams Work**

When you use:



Here’s what’s happening:

**1. Chunk-based reading and writing**

* fs.createReadStream() reads **small chunks** of the file at a time (default: 64 KB).
* It emits a 'data' event each time a chunk is ready.
* You can process each chunk as it's read.
* That chunk can be written using writeStream.write(chunk).

This means you're never loading the entire file into memory, just small parts of it — ideal for **large files**.

**📦 Default Stream Behavior**

**⚙️ Default chunk size (buffer size):**

| **Stream Type** | **Default HighWaterMark (Buffer Size)** |
| --- | --- |
| Readable | 64 KB for files |
| Writable | 16 KB |

The stream will read a chunk of this size, then wait for it to be processed (written or handled), then move to the next one. This ensures **backpressure** management (explained below).

**⚖️ Backpressure: Flow Control**

Streams automatically manage **flow control**:

* If the writable stream is slower than the readable stream, Node.js **pauses** reading until the write catches up.
* This avoids memory overflow and is a big advantage over manual reading.

**🔁 Streaming vs Normal fs.readFile() / fs.writeFile()**

| **Feature** | **fs.readFile() / fs.writeFile()** | **Streams (fs.createReadStream())** |
| --- | --- | --- |
| **Reads entire file?** | Yes (into memory) | No (reads in chunks) |
| **Memory usage** | High (depends on file size) | Low (controlled chunk size) |
| **Suitable for large files** | ❌ Risky / crashes | ✅ Efficient and safe |
| **Performance** | Slower with large files | Faster for large files |
| **Backpressure?** | ❌ No | ✅ Yes (automatically handled) |

**⏱️ At what rate do they read and write?**

Streams work **as fast as the system allows**, constrained by:

* The file system's read/write speed.
* CPU availability.
* Internal buffering (highWaterMark).
* Backpressure — if writing is slow, reading pauses.

You can control chunk size like this:



**🛑 Important Points**

* highWaterMark **does not set the chunk size** — it sets the **total buffer limit**.
* The actual size of chunks depends on the OS, file system, and stream implementation.
* When highWaterMark is exceeded, backpressure applies — the stream **pauses automatically**.

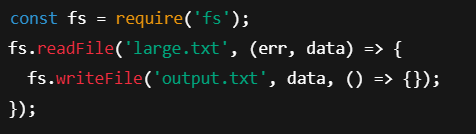
| **Question** | **Answer** |
| --- | --- |
| Is memory usage **equal** to highWaterMark? | ❌ Not always — it can be **slightly higher** |
| Is highWaterMark a **hard cap**? | ❌ No, it’s a **soft limit** |
| Can multiple streams multiply memory use? | ✅ Yes — each stream has its own buffer |
| Still better than fs.readFile()? | ✅ Absolutely — far lower and controlled |

**🧠 Example Scenario:**

**Using fs.readFile():**

Entire file read into memory before writing.

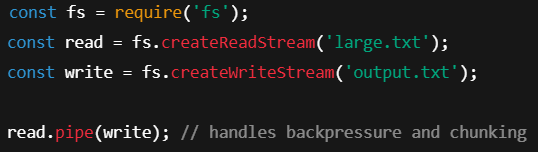
* Memory-heavy for large files.
* Can crash if file size > available memory.



**Using Streams:**

Efficient.

* Streams one chunk at a time.
* Automatically pauses/resumes as needed.



**🧠 Analogy**

Think of the stream as a **bucket** (highWaterMark) and the chunks of data as **cups of water** being poured in.

* The stream **keeps pouring** until the bucket is full.
* Once full, it **waits** (pauses reading) until some water is **poured out** (data consumed).
* Then it resumes filling again.

**🔁 Internally Workflow — Step-by-Step:**

**1. readStream reads data in chunks**

* Default chunk size (via highWaterMark) is 64 KB.
* As it reads each chunk, it calls writeStream.write(chunk).

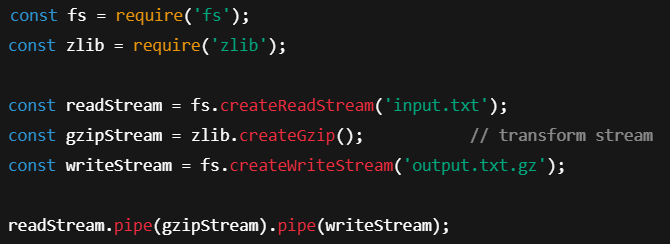
**2. If writeStream.write(chunk) returns false**

* It means: “⚠️ I can’t keep up — my buffer is full.”
* Node.js applies **backpressure**.
* readStream is **paused** — it stops reading new data.

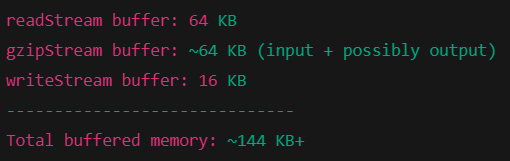
**3. Once writeStream drains its buffer**

* It emits a 'drain' event.
* readStream is **resumed** and continues reading more chunks.

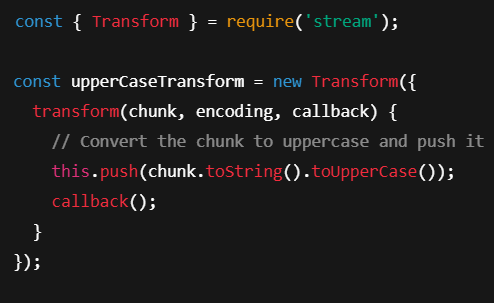
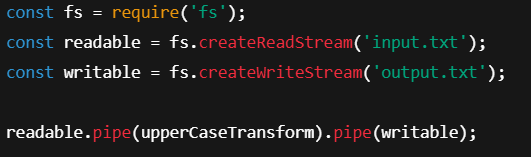
**🧱 Each stream has their own buffer**



* In this example, there are **three different stream objects**: readStream, gzipStream and writeStream
* Each of these streams is **independent** and has:
* Its own internal **buffer**
* Its own highWaterMark
* Its own logic for **flow control and backpressure**
* So, the memory in use could look like:



**🔧 Why use .pipe()?**

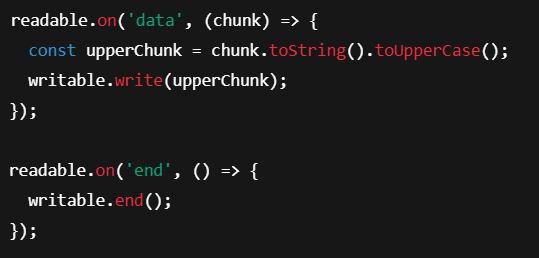
** **

This is a classic Node.js stream pipeline, and it's a chaining operation that connects three streams together: 

* **Efficient**: Pipes manage backpressure automatically (controls data flow rate).
* **Readable**: Much cleaner and less error-prone than manually handling on('data'), write(), etc.
* **Composable**: You can chain multiple pipes — e.g., .pipe(gzip).pipe(encrypt).pipe(writable)

**🔧 Bonus: Equivalent (Manual) Version Without .pipe()**

Just for understanding, here’s a rough manual version of what .pipe() is doing:



But with .pipe(), you don’t need to worry about flow control, buffering, or events — it handles everything.

**✅ Summary**

* **Streams read/write in chunks** – default is 64 KB for reading files.
* **Backpressure** prevents memory overload.
* **Much more efficient** than fs.readFile()/fs.writeFile() for large files.
* Let you **process or transform data** while reading.