§ THE NODEBOOK · CHAPTER 9

Foundation of *streams.*

Why streams exist, the push and pull models behind them, and how Node.js merges both into one abstraction for processing data that is too big to hold at once.

CHAPTER 9 · PUSH · PULL · BACKPRESSURE · PIPELINES

§ THE CORE QUESTION

Data bigger than *memory.*

Streams are not a Node.js invention. They answer a question as old as computing itself: how do you process data that does not fit in RAM, without waiting for all of it to arrive first?

Node.js streams are a direct response to physical memory limits and the realities of I/O. Not an arbitrary API choice.

---

§ THE OBVIOUS APPROACH

Three lines that don't *scale.*

Read the file, transform the buffer, write the result. Clean, easy to reason about, fine for small files. Then a user uploads 2GB.

JS

```
const data = await fs.readFile("input.mp4");
const processed = transform(data);
await fs.writeFile("output.mp4", processed);
```

§ FAILURE MODE

Read-everything fails *two ways at once.*

Memory is one half. The other half is shape: every step has to wait for the previous one to finish completely.

MEMORY	one 2GB file = 2GB allocated. Ten concurrent uploads = 20GB.
TIME	wait for the full read. then process. then wait for the full write.
NO OVERLAP	while bytes are still being read, nothing downstream can start.
SHAPE	read everything, then process everything, then write everything.

§ THE INSIGHT

Hold a chunk, not *the whole thing.*

You don't need the entire dataset in memory to process it. You only need the part you are working on right now. Pick a size (say 64 KB) and loop.

1. **read 64 KB**
   memory usage is bounded by chunk size, not file size
2. **process it**
   OS can already be reading the next chunk in the background
3. **write it**
   meanwhile the next chunk is being processed
4. **repeat until done**
   reading, processing, writing now overlap in time

§ DEFINITION

A stream is *chunk plumbing.*

Chunked processing introduces real complexity. Who reads next? What if the producer is faster than the consumer? How do you signal end, or clean up after an error mid-flow? A stream is the abstraction that handles all of that.

◆ WHAT A STREAM GIVES YOU

A structured way to read chunks, buffer them when needed, deliver them to your code, propagate end signals, and unwind cleanly on error.

§ TWO MODELS

Who decides when data *moves?*

Every streaming system commits to one answer. Push: the producer pushes when it has something. Pull: the consumer asks when it is ready. Different control flow, different problems.

	PUSH	PULL
who controls flow	producer	consumer
mechanism	emit events	call next()
fits	event-driven sources	on-demand sources
backpressure	must be added	implicit
fan-out	natural	needs extra work

§ PUSH ARCHITECTURE

Observer pattern, with a *data event.*

The push model is the Observer pattern from the 1994 Gang of Four book. The subject is the data source. The observers are the consumers. When new data is available, the subject notifies all of them.

IN NODE.JS

The core building block is `EventEmitter`. Subscribers register callbacks with `.on()`, the source calls `.emit()` when it has data.

§ PUSH, FROM SCRATCH

A push stream in *20 lines.*

Not production-ready. Just enough to see the mechanics. Take an array, emit each item as a `data` event, finish with `end`.

JS

```
import { EventEmitter } from "events";

class SimplePushStream extends EventEmitter {
  constructor(data) {
    super();
    this.data = data;
    this.index = 0;
  }

  start() {
    this._pushNext();
  }

  _pushNext() {
    if (this.index >= this.data.length) {
      this.emit("end");
      return;
    }
    const chunk = this.data[this.index++];
    this.emit("data", chunk);
    setImmediate(() => this._pushNext());
  }
}
```

§ THE CONSUMER SIDE

Subscribe and *react.*

The consumer doesn't ask for chunks. It registers listeners and lets the stream call them. This is the whole shape of push consumption.

JS

```
const stream = new SimplePushStream([1, 2, 3, 4, 5]);

stream.on("data", (chunk) => {
  console.log("Received:", chunk);
});

stream.on("end", () => {
  console.log("Stream ended");
});

stream.start();
```

§ A SMALL DETAIL

Why setImmediate *between chunks?*

Without it, `start()` runs a tight synchronous loop and emits every chunk before returning. The event loop can't process anything else. CPU is monopolized.

◇ YIELDING

Scheduling each `_pushNext` on a fresh tick lets timers, I/O callbacks, and other listeners run between chunks. It's a basic form of cooperative yielding, and you'll see this pattern repeatedly in async Node.js code.

§ FAILURE MODE

Pure push has no *brakes.*

If the producer is faster than the consumer, chunks pile up somewhere. The buffer grows without bound until the process runs out of memory.

THE GAP

Backpressure is the consumer telling the producer to slow down. Pure push has no channel for that. You have to add a separate contract on top: pause/resume signals, drain events, return values from write. It does not fall out of the model for free.

§ PULL ARCHITECTURE

Iterator protocol, one *next() at a time.*

Pull inverts the control flow. The consumer asks. Each `next()` returns `{ value, done }`. The producer only does work when called.

JS

```
class SimplePullStream {
  constructor(data) {
    this.data = data;
    this.index = 0;
  }

  next() {
    if (this.index >= this.data.length) {
      return { done: true };
    }
    return { value: this.data[this.index++], done: false };
  }
}

const stream = new SimplePullStream([1, 2, 3, 4, 5]);
let result = stream.next();
while (!result.done) {
  console.log("Pulled:", result.value);
  result = stream.next();
}
```

§ SYNTACTIC SUGAR

Generators write *iterators for you.*

A `function*` pauses on `yield` and resumes on the next `next()`. The runtime gives you a working iterator (and an iterable) for free, so `for ... of` just works.

JS

```
function* simplePullStream(data) {
  for (const chunk of data) {
    yield chunk;
  }
}

for (const chunk of simplePullStream([1, 2, 3, 4, 5])) {
  console.log("Pulled:", chunk);
}
```

§ ASYNC PULL

for await...of for *real I/O.*

Files, sockets, and queries are async. An async iterator's `next()` returns a Promise. Standardized in ES2018. Node.js Readable streams support it natively.

JS

```
async function* asyncPullStream(data) {
  for (const chunk of data) {
    await new Promise((resolve) => setImmediate(resolve));
    yield chunk;
  }
}

for await (const chunk of asyncPullStream([1, 2, 3])) {
  console.log("Pulled:", chunk);
}
```

§ PULL, THE TRADE-OFFS

Free backpressure, *awkward fan-out.*

Pull buys you flow control by construction, but it pays for it elsewhere. The model fits on-demand sources well and external events poorly.

BACKPRESSURE	implicit. slow consumer = slow producer.
LAZINESS	producer does no work until asked, supports infinite sequences
COMPOSITION	pipelines drive backward from the final consumer
EVENT SOURCES	can't pull data that hasn't arrived yet
FAN-OUT	pulled values are consumed once, broadcast needs extra work

§ NODE.JS STREAMS

Push at the core, pull on the *surface.*

Node.js streams pick the push model as their foundation, then bolt on explicit backpressure and async iteration so you can consume them either way.

BASE	extends EventEmitter, data flows via "data" events
FLOW CONTROL	consumer can pause, write() returns false when buffer is full
PULL SURFACE	Readable streams work with for await...of out of the box
COST	a richer API surface and a few sharp edges that come with combining models

§ HOW WE GOT HERE

Streams shipped, then *evolved.*

Streams have been in Node.js from the start. The shape of today's API is the result of two big rewrites that fixed real problems with the previous one.

1. **pre-0.10: Streams1**
   data events only. no pause, no backpressure. slow consumer = unbounded buffering.

2. **0.10: Streams2**
   paused vs flowing modes. read() pulls. write() returns false. drain event signals readiness.

3. **v10 and later**
   stream.pipeline() for clean error handling, stream.finished() for completion, async iterator support.

§ VOCABULARY

Four stream *types.*

Each type is a role in the data flow. Combine them and you get pipelines: source, optional transforms, sink.

TYPE	ROLE	EXAMPLES
Readable	source of data	fs.createReadStream, http.IncomingMessage, process.stdin
Writable	sink for data	fs.createWriteStream, http.ServerResponse, process.stdout
Transform	consumes and produces	zlib.createGzip, crypto.createCipheriv
Duplex	two independent sides	net.Socket (TCP connection)

§ DATA FORWARD, PRESSURE BACKWARD

Pipelines self- *regulate.*

Chunks move from source to sink. When the sink's buffer fills, it signals upstream and the whole pipeline pauses until the buffer drains. Memory stays bounded.

01

READABLE

reads chunks from a file

02

TRANSFORM

uppercases each chunk

03

WRITABLE

writes to another file

◆ WHY THIS MATTERS

Without backpressure, fast stages outrun slow stages and buffers grow forever. With it, the pipeline runs at the speed of the slowest stage.

§ WHEN STREAMS ARE RIGHT

Not always the right *tool.*

Streams add overhead: scheduled callbacks, chunking, backpressure bookkeeping. For a 10 KB JSON file, just call `fs.readFile`. Reach for streams when one of these is true.

BIG OR UNBOUNDED DATA	multi-GB log files, HTTP bodies of unknown size, long-lived sockets
START BEFORE YOU HAVE IT ALL	send first bytes of a download as they're read, lower time to first byte
TRANSFORMATION PIPELINES	parse, filter, map, aggregate as composable stages
REAL-TIME FEEDS	message queues, IoT sensors, user events. handle each as it arrives.
PROXY / MULTIPLEX	pipe between sockets without buffering whole messages