Readable *streams.*

A Readable stream sits between a source and a consumer with an internal buffer in the middle. This chapter covers the class, its events, paused vs flowing modes, internal buffering, highWaterMark, custom _read implementations, every consumption pattern, and where backpressure quietly fails.

CHAPTER 10

READABLE · HIGHWATERMARK · BACKPRESSURE · _READ

§ ENTRY POINT

Readable streams *produce data.*

Files, sockets, generators, db rows. The Readable class is what sits between a producing source and your consumer code, with an internal buffer, two operating modes, and an event-driven contract.

§ THE CLASS

Readable extends *EventEmitter.*

Every Readable is an event emitter. Most of its public contract is expressed through events. You rarely construct one directly, fs.createReadStream and http.IncomingMessage hand them to you, but the options matter either way.

JS

```
import { Readable } from "stream";

const readable = new Readable({
  highWaterMark: 1024, // 1KB buffer
});
```

§ OPTION

highWaterMark is the *buffer threshold.*

Max bytes (or objects) the stream will hold internally before it stops asking the source for more. When buffer length is below it, _read is called to refill. At or above it, the stream waits for the consumer to drain.

DEFAULT (BYTE MODE)	`65536 bytes (64 KB)`
DEFAULT (OBJECTMODE)	`16 objects`
RULE	`if state.length ≥ highWaterMark, push() returns false`
MEANING	`a soft cap on memory, not a hard limit`

§ OPTION

objectMode buffers *objects, not bytes.*

Default Readable streams emit Buffers and strings. Flip objectMode and chunks become arbitrary JS values. highWaterMark switches from bytes to a count of objects.

JS

```
const objectStream = new Readable({
  objectMode: true,
  highWaterMark: 100, // buffer up to 100 objects
});
```

§ SURFACE AREA

Five events you *actually care about.*

Readable streams talk to you through events. Most of your interaction will revolve around these five and what they imply about state.

EVENT	MEANING
data	chunk available, flowing mode pushes it to you
end	source is exhausted, no more data coming
error	something blew up, attach a listener or it throws
readable	paused mode signal that read() will return non-null
close	underlying resources released (distinct from end)

§ SIDE EFFECT

Attaching data flips *the stream on.*

The act of subscribing to data switches the stream into flowing mode. Data starts arriving as soon as it's available. You don't pull, the stream pushes.

JS

```
readable.on("data", (chunk) => {
  console.log(`Received ${chunk.length} bytes`);
});
```

§ PULL PATTERN

readable means *data is buffered.*

Paused mode signal. Loop read() inside the handler until it returns null. You decide when to pull instead of taking whatever the source pushes.

JS

```
readable.on("readable", () => {
  let chunk;
  while ((chunk = readable.read()) !== null) {
    console.log(`Read ${chunk.length} bytes`);
  }
});
```

§ PITFALL

Always attach an *error listener.*

File deleted mid-read, network drop, source throws. The stream emits error with the error object. With no listener attached, Node throws and the process can crash.

✕ PITFALL

Unhandled error events on a stream propagate up and can take the whole process down. Attach a listener even if it just logs.

§ TWO MODES

Paused pulls, *flowing pushes.*

Every Readable is in one of two modes. The mode controls whether you call read() or whether data events fire on their own. New streams start paused.

MODE	BEHAVIOR
paused	buffer fills to highWaterMark, you call read() to drain
flowing	data emits as soon as a chunk lands in the buffer

§ TRANSITIONS

What flips a stream *into flowing.*

Three actions move a stream from paused to flowing. One method moves it back, but only if it isn't piped. pipe() owns its own flow control.

1. paused -> flowing: attach a data listener
2. paused -> flowing: call resume()
3. paused -> flowing: call pipe(writable)
4. flowing -> paused: call pause(), only when not piped

§ INTERNALS

The buffer is an *array of chunks.*

Not a single big Buffer. An array of the original allocated chunks tracked in order. Used to be a linked list, switched for cache locality. _readableState exposes the current state, useful for debugging.

JS

```
const state = readable._readableState;
console.log(`Buffer length: ${state.length} bytes`);
console.log(`Buffer count: ${state.buffer.length} chunks`);
console.log(`highWaterMark: ${state.highWaterMark} bytes`);
```

§ CONTRACT

_read fetches, *push delivers.*

When the buffer dips below highWaterMark, the stream calls _read(size). Your job: fetch from the source and push(chunk). push(null) signals end. push returns false when the buffer hits the threshold, telling you to stop.

JS

```
class MyReadable extends Readable {
  _read(size) {
    const chunk = this.getDataFromSomeTypeOfSource(size);
    if (chunk) {
      this.push(chunk); // adds to internal buffer
    } else {
      this.push(null); // signals end of data
    }
  }
}
```

§ MINIMAL EXAMPLE

Counter stream: *extend, _read, push.*

Numbers 1 to N as strings. No need to check push's return here. Source is synchronous and the stream only calls _read when the buffer has space.

JS

```
class CounterStream extends Readable {
  constructor(max, options) {
    super(options);
    this.max = max;
    this.current = 1;
  }

  _read() {
    if (this.current <= this.max) {
      this.push(String(this.current));
      this.current++;
    } else {
      this.push(null);
    }
  }
}
```

§ REALISTIC EXAMPLE

A line-splitting *readable stream.*

Reads bytes, accumulates into a string buffer, pushes complete lines. Notice the push() return check inside the while loop. If it returns false, we stop pushing and let _read get called again later.

JS

```
class LineStream extends Readable {
  constructor(filePath, options) {
    super(options);
    this.fd = fs.openSync(filePath, "r");
    this.buffer = "";
    this.position = 0;
  }

  _read() {
    const chunk = Buffer.alloc(1024);
    const bytesRead = fs.readSync(this.fd, chunk, 0, 1024, this.position);
    if (bytesRead === 0) {
      if (this.buffer.length > 0) this.push(this.buffer);
      this.push(null);
      return;
    }
    this.position += bytesRead;
    this.buffer += chunk.slice(0, bytesRead).toString();
```

§ PATTERN TO AVOID

data + async handler = *unbounded concurrency.*

The handler is async, but the stream doesn't wait. Each chunk fires another fetch. With a large file and small chunks you get thousands of in-flight requests, then memory and sockets run out.

JS

```
const readable = fs.createReadStream("large-file.txt");

readable.on("data", async (chunk) => {
  await fetch("https://api.example.com/process", {
    method: "POST",
    body: chunk,
  });
});

readable.on("end", () => {
  console.log("Done");
});
```

§ THE FIX

Manual pause/resume *works, awkwardly.*

Pause before the await, resume after. Serializes processing, the file read rate matches the request rate. Works, but if any error skips the resume the stream gets stuck.

JS

```
readable.on("data", async (chunk) => {
  readable.pause();
  await fetch("https://thenodebook.com/process", {
    method: "POST",
    body: chunk,
  });
  readable.resume();
});
```

§ BETTER FIX

for await..of gives you *backpressure for free.*

The loop pulls the next chunk only after the body resolves. Producer rate matches consumer rate without a single pause/resume call. Recommended path for most stream consumption.

JS

```
const readable = fs.createReadStream("large-file.txt");

for await (const chunk of readable) {
  await fetch("https://thenodebook.com/process", {
    method: "POST",
    body: chunk,
  });
}

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

§ REFERENCE CARD

Consumption patterns *and what they enforce.*

Backpressure is not automatic. Whether you get it depends on how you consume. Pick based on whether the work is fast and synchronous or slow and async.

PATTERN	BACKPRESSURE
`data + end listeners`	`manual (pause/resume)`
`for await...of`	`automatic`
`readable + read()`	`manual, you control pulls`
`read(size) for fixed headers`	`manual, non-blocking`
`pipe(writable)`	`automatic, errors do not propagate`
`stream.pipeline`	`automatic, errors and cleanup handled`

§ COMPOSITION

pipeline cleans up *when something fails.*

pipe() handles flow control but errors don't propagate and cleanup is on you. pipeline from stream/promises returns a promise, destroys every stream on error, and closes resources. Use this in production.

JS

```
import { pipeline } from "stream/promises";

try {
  await pipeline(readable, writable);
  console.log("Pipeline succeeded");
} catch (err) {
  console.error("Pipeline failed:", err);
}
```

§ STATE MACHINE

readableFlowing has *three values.*

null on creation. true once data flows. false only after an explicit pause. Removing all data listeners does not pause, the stream keeps draining its source into the void.

READABLEFLOWING	MEANING
null	fresh stream, never started
true	flowing, data events fire to whoever is listening
false	paused explicitly via pause() or pipe backpressure

§ OBJECT MODE

Streaming database *rows as objects.*

Each row is a chunk. The buffer count controls when _read pauses, which naturally throttles the next batch query. No JSON serialization round trip.

JS

```
class RowStream extends Readable {
  constructor(db, query, options) {
    super({ ...options, objectMode: true });
    this.db = db;
    this.query = query;
    this.offset = 0;
  }

  async _read() {
    try {
      const rows = await this.db.query(this.query, {
        offset: this.offset, limit: 100,
      });
      for (const row of rows) this.push(row);
      rows.length > 0 ? (this.offset += rows.length) : this.push(null);
    } catch (err) {
      this.destroy(err);
    }
  }
}
```

§ PITFALL

objectMode counts *objects, not bytes.*

Node has no way to measure the byte size of arbitrary JS objects. The accounting is purely a count. You decide a sane highWaterMark based on what each object actually weighs.

⚠ WARNING

16 objects of 10 MB each = 160 MB buffered. The default of 16 looks tiny but can hold a lot of memory if your objects are heavy. Set highWaterMark accordingly.

§ TUNING

Choosing a value for *highWaterMark*.

64 KB is a balance between memory and syscall overhead, the Node team picked it through experimentation. The default usually works. Tune when profiling tells you to.

SCENARIO	DIRECTION
fast SSD, big files, throughput matters	larger (128 KB or 256 KB)
thousands of concurrent streams in a server	smaller (4 KB) to cap per-stream RAM
real-time / live feed	smaller, less buffering = lower latency
objectMode with heavy objects	lower count to bound memory

§ SHORTCUT

Readable.from wraps any *iterable as a stream.*

Async generator, sync array, regular generator. Readable.from drives next(), pushes the value, signals end when the iterator is done. No need to extend the class.

JS

```
async function* generateNumbers() {
  for (let i = 1; i <= 5; i++) {
    await new Promise((resolve) => setTimeout(resolve, 100));
    yield i;
  }
}

const stream = Readable.from(generateNumbers());

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

§ EDGE CASES

Things that bite you *at the boundaries.*

Each one comes from a stream being in a state you didn't expect. When debugging, log _readableState and start there.

1. empty stream: push(null) with no prior data, end fires, data never does
2. destroy() drops buffered data, use push(null) for graceful end
3. mixing readable + data listeners: readable wins, data may not fire
4. read(size) is non-blocking, returns whatever is available or null
5. check readable.destroyed before calling read or push from custom code

§ RECAP

One question to *hold onto.*

If you can answer this without thinking, the rest of the streams chapter clicks into place.

Q

In flowing mode, what makes the stream stop calling _read on the source?

TAP TO REVEAL