§ THE NODEBOOK · CHAPTER 5

What is *a Buffer?*

Server-side data isn't text, it's bytes. Buffers are Node's view onto raw memory: how they're laid out, how to encode and decode them, and why they live outside the V8 heap.

CHAPTER 5

BYTES · HEX · ENCODINGS

§ THE MISMATCH

Server-side data isn't *text.*

JavaScript grew up around HTML, JSON, and Unicode strings. Node moved it to filesystems, sockets, and crypto, where the unit is a raw byte with no encoding attached.

⚠ WARNING

A TCP packet, a JPEG header, a PNG file. None of them are text. Reaching for a string to hold them is the first wrong move.

§ BASICS

Bit, byte, *0 to 255.*

A bit is a single 0 or 1. Eight bits grouped together is a byte. With 8 bits, the smallest value is 0 and the largest is 255. Every byte you ever touch sits in that range.

BIT	one 0 or 1, smallest unit of data
BYTE	8 bits, one of 256 possible patterns
VALUE RANGE	0 to 255, always
WHY IT SHOWS UP	RGB channels, buf[i], any byte API

§ INTERPRETATION

A byte has no *meaning.*

The pattern `01001001` is just a pattern. It is not the letter I, not the color blue, not a sample of audio. Code applies the interpretation. The byte does not carry one.

INFO

Every binary data bug comes from applying the wrong interpretation. Pick the right encoding, the right struct layout, the right endianness, and the byte makes sense. Pick the wrong one and you get garbage.

§ ONE BYTE, MANY MEANINGS

Same bits, *different stories.*

The byte `01001001` is the number 73. ASCII reads 73 as the letter I. A PNG decoder might read it as a blue channel intensity. A CPU might treat it as an opcode.

CONTEXT	INTERPRETATION OF 01001001
integer (base-2)	73
ASCII / UTF-8	the character 'I'
image pixel channel	intensity 73 of 255
audio sample	speaker position 73
machine code	one opcode for the CPU

§ NOTATION

Hex is the byte *shorthand.*

One hex digit covers 4 bits. Two hex digits cover a full byte. That is why Node prints buffers in hex. Same bytes, easier to read.

```
"HELLO" as bytes:

binary    01001000 01000101 01001100 01001100 01001111
hex       48       45       4C       4C       4F
char      H        E        L        L        O

console.log(Buffer.from("HELLO"))
// <Buffer 48 65 6c 6c 6f>
```

§ MULTI-BYTE NUMBERS

Endianness is the *byte order.*

A 16-bit number is two bytes. Stored most-significant first is big-endian. Least-significant first is little-endian. Networks tend to be BE, x86 CPUs are LE. You have to tell Node which one to use.

```
value 0x1234 stored in memory:

big-endian      [ 0x12 ][ 0x34 ]   // network byte order
little-endian   [ 0x34 ][ 0x12 ]   // x86, ARM default

buf.readInt16BE(0)  // pick the BE rule
buf.readInt16LE(0)  // pick the LE rule
```

§ PATTERN TO AVOID

Reading a PNG *as utf8.*

The naive file copy. Read the file as a string, write it back. It looks fine until you check the output file size and try to open it.

NAIVE-COPY.JS

JS

```
import fs from "fs";

const data = fs.readFileSync("logo.png", "utf8");
console.log(data.slice(0, 50));

fs.writeFileSync("logo-corrupted.png", data);
```

§ FAILURE MODE

The replacement *character trap.*

PNG starts with bytes `89 50 4E 47`. UTF-8 sees `0x89` and has no rule for it. The decoder cannot crash, so it emits U+FFFD and discards the original byte.

```
PNG header bytes:   89 50 4E 47

utf8 decoder reads 0x89:
  - not a valid start byte in UTF-8
  - emits U+FFFD (the <garbled> char)
  - original 0x89 is gone

writeFileSync now stores EF BF BD instead.
```

✕ PITFALL

U+FFFD is a one-way conversion. The original byte is replaced by the three UTF-8 bytes for the replacement char. The file gets bigger or smaller depending on inputs, and the data is permanently corrupted.

§ PATTERN TO AVOID

latin1 round-trip is a *fragile hack.*

latin1 maps bytes 0-255 one-to-one to the first 256 Unicode code points, so the round trip can preserve bytes. The result is still a JS string, and V8 may apply text-only optimizations to it.

⚠ WARNING

You are lying to the runtime about what the data is. It is binary, not European linguistic text. Pass that string to any API that expects real text and you can get silent corruption later. Stop decoding bytes into strings, hold the bytes directly.

§ WHY NOT ON THE V8 HEAP

500MB of bytes *freezes the GC.*

V8 is tuned for many small interconnected JS objects. A single 500MB byte array on the V8 heap means major GCs have to scan and possibly relocate that whole block. The main thread stalls for seconds.

⚠ WARNING

V8 compacts memory by moving live objects. Moving half a gigabyte during a stop-the-world pause is the kind of latency that takes a server down. Big static blobs need a different home.

§ ARCHITECTURE

Two heaps, one *handle.*

Buffer data is allocated outside the V8 heap by Node's C++ core. The JS-side `Buffer` object is a small handle that holds a pointer plus length. The handle lives on the V8 heap, the bytes do not.

RAW SLAB	contiguous bytes in C++ memory, off-heap
JS HANDLE	tiny object on V8 heap, holds pointer + length
GC SEES	just the handle, never walks the slab
ON COLLECT	C++ layer notified, slab freed back to OS

§ HISTORY

Why Node shipped its own *byte type.*

Node started in 2009. ArrayBuffer and TypedArrays were experimental proposals at the time, not stable in V8. Node needed binary I/O on day one for HTTP, sockets, and the filesystem.

◆ INFO

Buffer was the pragmatic answer. A fixed-size mutable byte sequence with an ergonomic, server-focused API. Years later, when TypedArray stabilized, Buffer was rebased on top of it.

§ ALLOCATION

alloc zeros, *allocUnsafe doesn't.*

`Buffer.alloc(size)` hands back a zero-filled buffer. `Buffer.allocUnsafe(size)` skips the fill. Faster, but the bytes can be whatever was sitting in that memory region.

JS

```
const buf1 = Buffer.alloc(10);
console.log(buf1);
// <Buffer 00 00 00 00 00 00 00 00 00 00>

const buf2 = Buffer.allocUnsafe(10);
// contents are uninitialized memory
```

✕ PITFALL

allocUnsafe leaks whatever the OS handed you. Could be zeros, could be a chunk of an old request body holding a session token. Only use it if you immediately overwrite every byte.

§ CONSTRUCTION

Buffer.from for *existing data.*

`Buffer.from` takes a string with an encoding, an array of byte values, or another buffer. The string form is the correct inverse of the broken read-as-utf8 we saw earlier.

BUFFER-FROM.JS

JS

```
const a = Buffer.from("hello world", "utf8");
// <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>

const b = Buffer.from([0x68, 0x65, 0x6c, 0x6c, 0x6f]);
// <Buffer 68 65 6c 6c 6f>
b.toString("utf8");  // "hello"

const c = Buffer.from(a);   // copy of a
c[0] = 0x78;
c.toString("utf8");  // "xello world"
a.toString("utf8");  // "hello world"
```

§ ACCESS

Index in, toString *out.*

Read and write a single byte by index. Convert to text safely with `toString` and an encoding. This is representation, not the destructive decode that mangled the PNG.

JS

```
const buf = Buffer.from("hey");
buf[0];       // 104  (ASCII 'h')
buf[1] = 0x6f;
buf.toString("utf8");  // "hoy"

const s = Buffer.from("my-secret");
s.toString();          // "my-secret"
s.toString("hex");     // "6d792d736563726574"
s.toString("base64");  // "bXktc2VjcmV0"
```

§ WRITE

buf.write places *bytes at offset.*

`buf.write` writes a string into a buffer at an offset and returns the number of bytes written. Useful when you are assembling a binary frame by hand, like an HTTP response header.

JS

```
const out = Buffer.alloc(128);

let offset = out.write("HTTP/1.1 200 OK\r\n");
offset += out.write("Content-Type: text/plain\r\n", offset);

console.log(out.toString("utf8", 0, offset));
// HTTP/1.1 200 OK
// Content-Type: text/plain
```

§ INHERITANCE

Buffer is a *Uint8Array subclass.*

Since Node 3, `Buffer` extends `Uint8Array`. Any API that accepts a Uint8Array accepts a Buffer. You get the web standard plus Node's server-side methods on top.

JS

```
const buf = Buffer.alloc(10);

buf instanceof Buffer;      // true
buf instanceof Uint8Array;  // true
```

◆ INFO

Standard Uint8Array methods (slice, subarray, map, filter) work on Buffers. The Node-only methods (toString hex, write, readInt16BE) only exist on Buffer.

§ MEMORY MODEL

ArrayBuffer is the *raw slab.*

`ArrayBuffer` is the byte storage. It has no read or write API of its own. `Buffer`, `Uint8Array`, `Int16Array`, and the rest are views that decide how to interpret those bytes.

ARRAYBUFFER	the bytes themselves, no methods
UINT8ARRAY VIEW	reads/writes as 8-bit unsigned ints
INT16ARRAY VIEW	reads/writes as 16-bit signed ints
BUFFER VIEW	Uint8Array plus Node helpers
BUF.BUFFER	gives you the underlying ArrayBuffer

§ SHARED MEMORY

Multiple views, one *ArrayBuffer.*

Two views over the same ArrayBuffer see the same bytes. Write through one, read through the other. No copy. Useful for parsing structs of mixed widths in place.

SHARED-MEMORY.JS

JS

```
const ab = new ArrayBuffer(4);
const u8 = new Uint8Array(ab);
const i32 = new Int32Array(ab);

u8[0] = 0xff;
u8[1] = 0xff;
u8[2] = 0xff;
u8[3] = 0x7f;

i32[0];  // 2147483647 on a little-endian host
```

§ RECAP

One sentence to *hold onto.*

Strings can't safely hold raw bytes. V8's heap can't cheaply hold large ones. Buffer answers both with an off-heap slab and a thin JS handle, and it sits on top of the standard Uint8Array.

A Node.js Buffer is a server-side-ergonomic subclass of Uint8Array, a view over a raw block of memory allocated outside the V8 garbage-collected heap.
