MENTORMATE™
DevCamp

Building Real-Time Applications

Yulia Tenincheva

Senior Cloud Engineer, MentorMate

Today's Agenda

- Buffers
- Streams
- WebSockets

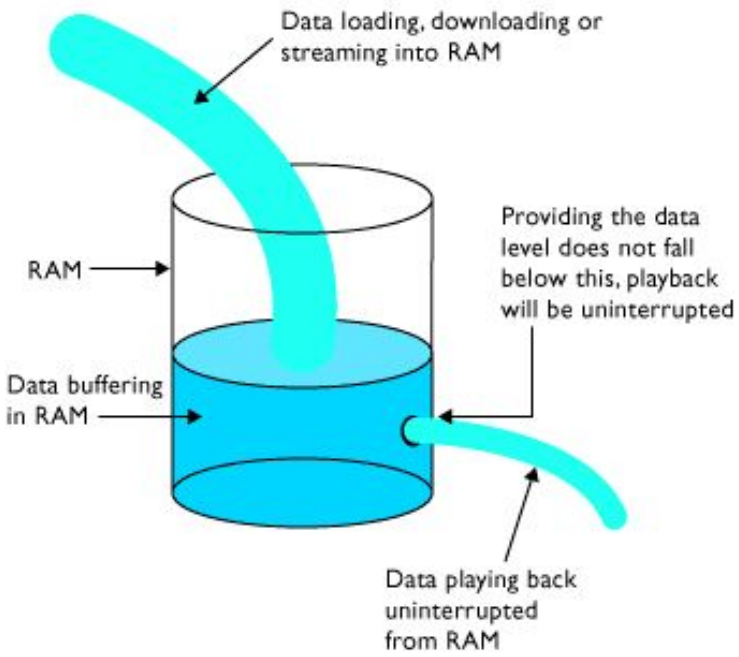
Buffers

What is a Buffer?

- Any data read (written) to (from) a file (network) is read (written) in **Bytes**.

- Buffer objects are used to represent a fixed-length sequence of bytes. To work with binary data, we need access to these buffers.

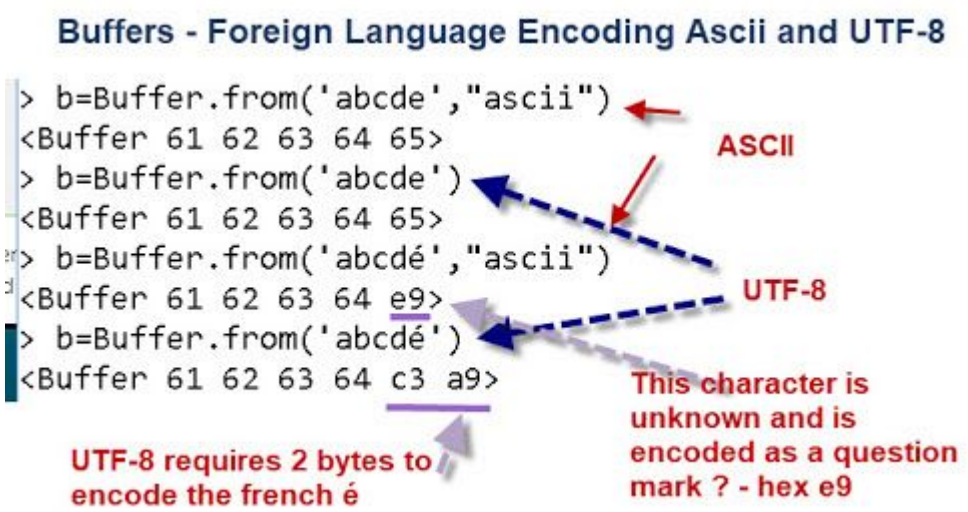
<Buffer ff f3 44 c4 00 11 88 96 20 00 ca 52 4c 03 e0 6d c6 4c 53 75 7b e4 b0 f4 22 1f 99 04 7a 33 86 43 6a 3e 94 64 13 44 61 7a 6d 36 85 05 cb 19 27 83 07 de ... 4270 more bytes>



Encoding Demo

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  console.log(buf.toString());
});
```

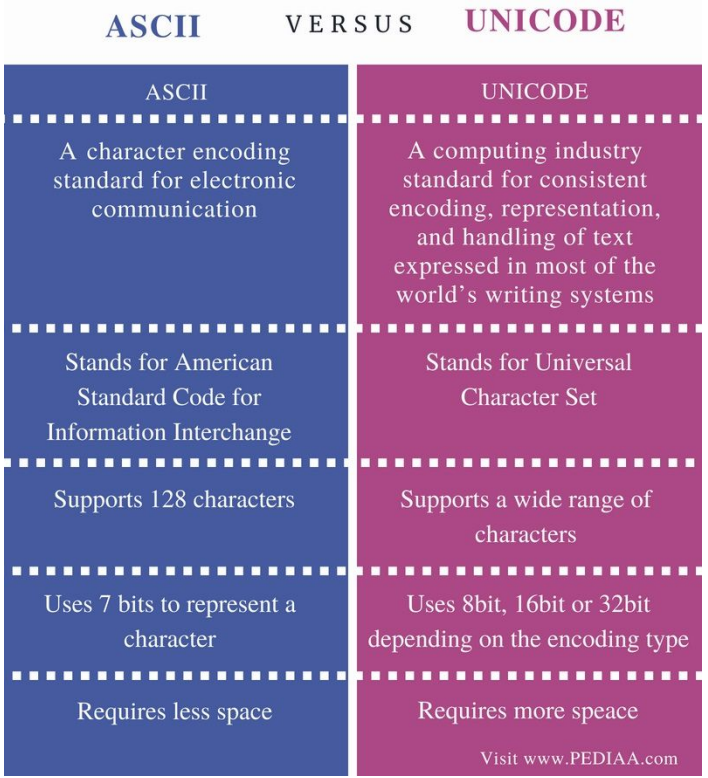
toString by default will convert data into a UTF-8 encoded string.



- Supported by Node.js:
  - Character:
    - `utf8`
    - `utf16le`
    - `latin1`
    - `ascii`
  - Binary-to-text:
    - `base64`
    - `hex`

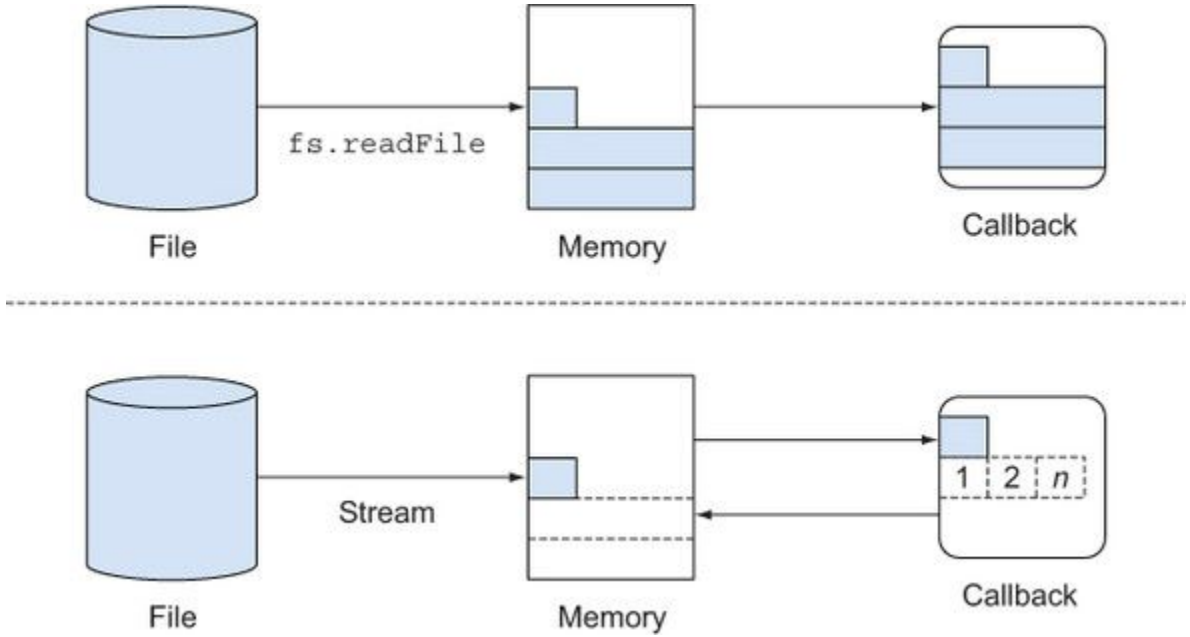
Resources

- Buffers in Node.js - official API documentation
- Buffers: Working with bits, bytes, and encodings - chapter of a book.
- Why buffers matter - blog post
- Typed arrays in JavaScript - docs



Streams

Buffers vs Streams



Using streamable APIs means I/O operations potentially use less memory.

Streams make programming in node **simple**, **elegant** and **composable**.

—James Hillday

Why Streams?

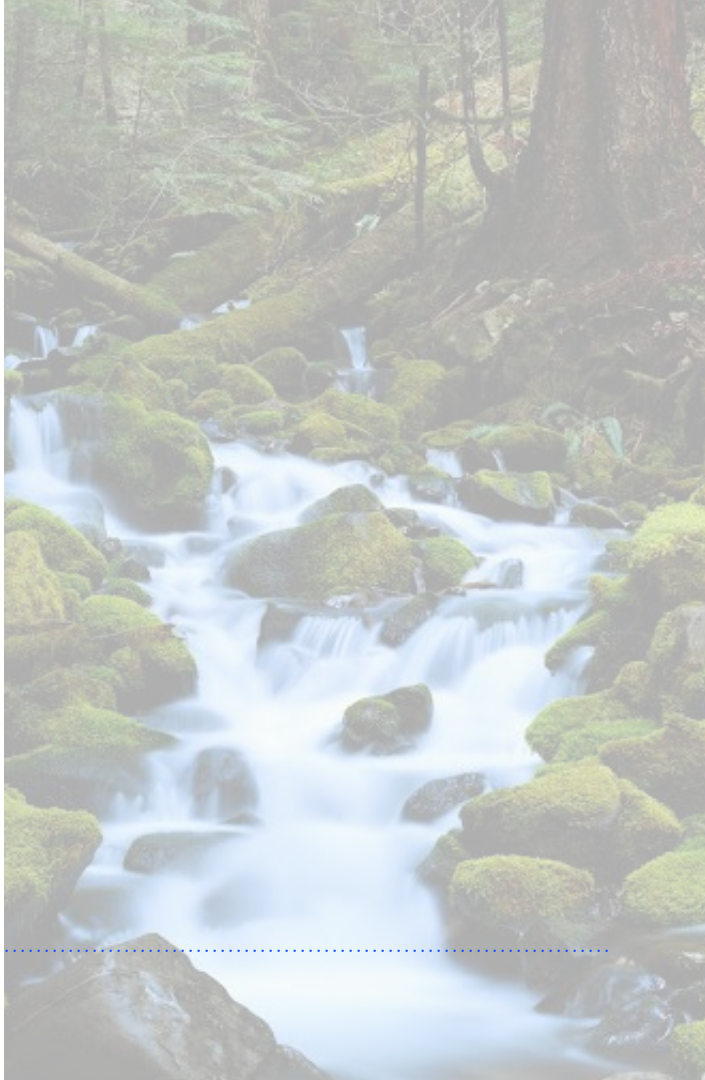
- Allows for data processing chunk-by-chunk or line-by-line
- Memory efficient - do not need to save entire large file/buffer to memory
- Allows for more functional approach - small modules can be chained
- Streams are essential for scalability

How streams work?

- Streams are Event Emitters
- Streams are the basic I/O of node processes
- Writable streams must send a signal back to the readable streams that they are ready for more data.

```
// readable process.stdin is directly echoed back to process.stdout

process.stdin.pipe(process.stdout);
```



Bestiary of Streams

- **Readable** - can act like the source, but not the destination
- **Writable** - can act as the destination, but can not be a source
- **Duplex** - both Readable and Writable
- **Transform** - between Readable and Writable

Readable Streams	Writable Streams
HTTP responses, on the client	HTTP requests, on the client
HTTP requests, on the server	HTTP responses, on the server
fs read streams	fs write streams
zlib streams	zlib streams
crypto streams	crypto streams
TCP sockets	TCP sockets
child process stdout and stderr	child process stdin
process.stdin	process.stdout, process.stderr

Bestiary of Streams

- **Readable** - can act like the source, but not the destination
- **Writable** - can act as the destination, but can not be a source
- **Duplex** - both Readable and Writable
  - **Transform** - between Readable and Writable

Readable Streams

Events

- data
- end
- error
- close
- readable

Functions

- pipe(), unpipe()
- read(), unshift(), resume()
- pause(), isPaused()
- setEncoding()

Writable Streams

Events

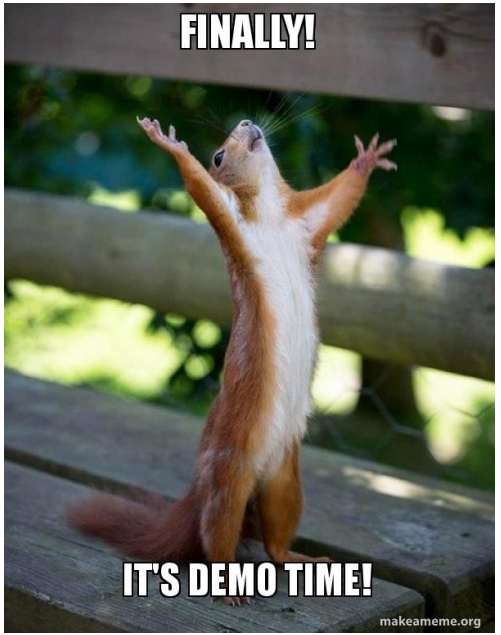
- drain
- finish
- error
- close
- pipe/unpipe

Functions

- write()
- end()
- cork(), uncork()
- setDefaultEncoding()

Resources & Demos

- Node's most powerful and misunderstood feature - Streams (chapter from a book)
- Everything you should know about streams - blog post
- Demos:
  - Buffer vs Streams - `fs.readFile` vs `fs.createReadStream`
  - Pipe example, `pipe` to server response
  - Encrypt/Decrypt Transform stream example



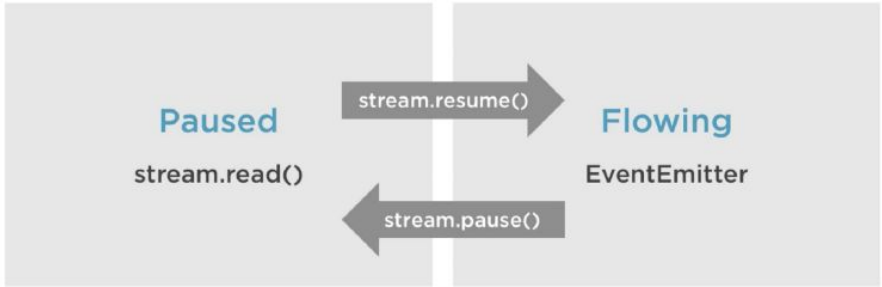
Readable Streams

- All readable streams start in the **paused mode by default**. One of the ways of switching the mode of a stream to flowing is to attach a ‘data‘ event listener.
- A way to switch the readable stream to a flowing mode manually is to call the **stream.resume** method. If there are no handlers, the data is **lost**.

```
const fs = require('fs');
const stream = fs.createReadStream('./small.txt');

stream.resume();

setTimeout(() => {
  stream.on('data', (data) => {  console.log(data);  });
}, 2000);
```

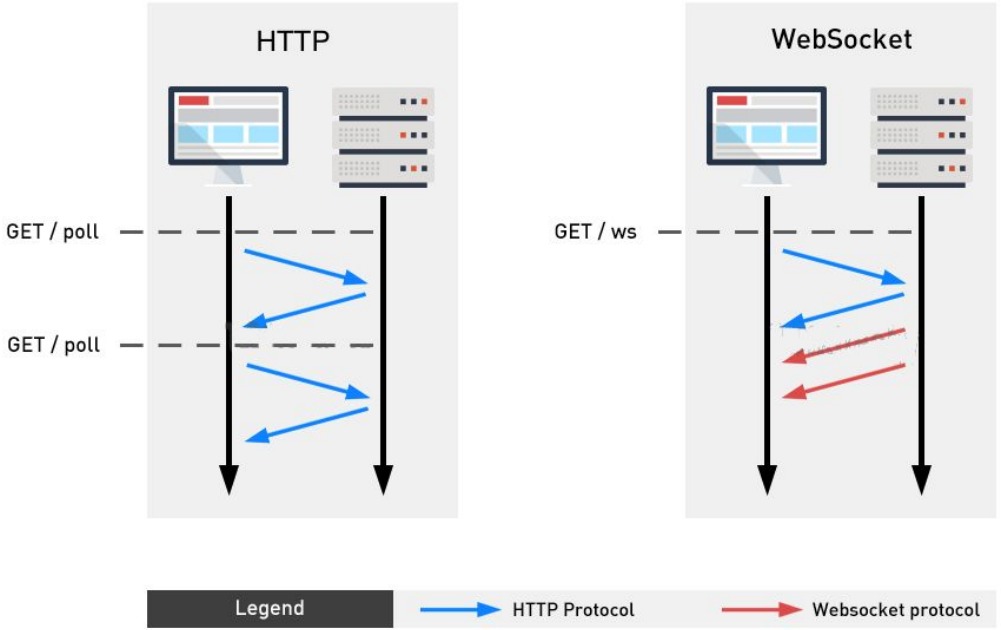


Demo: Write a custom Readable stream

WebSockets

WebSockets

- WebSockets is a **bidirectional** communication **protocol** over the web
- **Reduces latency**, saves bandwidth and CPU power
- Provides **enhanced capabilities** to HTML5 applications



How WebSockets work?

- Establish connection, open handshake
  WebSocket URIs use a new scheme ws:
  (or wss: for a secure WebSocket)

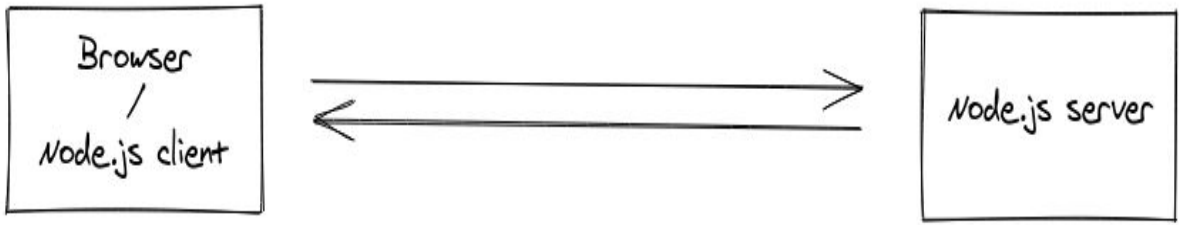
```
"ws:" "//" host [ ":" port ] path [ "?" query ]
"wss:" "//" host [ ":" port ] path [ "?" query ]
```

- WebSocket connections are established by ***upgrading*** an HTTP request

- WebSocket is a *framed* protocol, meaning that a chunk of data (a message) is divided into a number of discrete chunks, with the size of the chunk encoded in the frame.

Socket.IO

- **Socket.IO** provides **additional features** over a plain WebSocket object. It is NOT a WebSocket implementation!
- Socket.io NPM Package
- Emit cheatsheet
- Client installation



Build a Chat Application

Homework