Streams

Hungry? Why wait? (for the rest of your data)

Streams: Intro

"We should have some ways of connecting programs like garden hose--screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also."

-- Doug McIlroy. October 11, 1964
Original developer of Unix pipes

Streams: Intro

Streams are like Arrays, but laid out in time, rather than in memory.

Streams: Intro

Streams are useful for:

- Start processing data before the last bit is received
  - Start sending a response before the request is completely received
- Process data in pieces without buffering the whole
  - Especially useful for processing large files too big to fit in memory
- Normalized API for handling disparate data types

Streams: Types

There are 5 kinds of streams:

- readable
- writable
- transform
- duplex
- “classic”

Streams: Commonalities (.pipe())

All streams use *.pipe()* to pair inputs with outputs.

.pipe() connects a readable stream's output to a writable stream's input

```
1 // .pipe() chaining
2 a.pipe(b).pipe(c).pipe(d)
3
4 // Equivalent to
5
6 a.pipe(b)
7 b.pipe(c)
8 c.pipe(d)
```

Streams: Readable

A readable stream is a stream that outputs data.

Streams: Readable

A readable stream is a stream that **outputs** data.

Examples of readable streams:

- http responses, on the client
- http requests, on the server
- fs read streams
- zlib streams
- crypto streams
- tcp sockets
- child process stdout and stderr
- process.stdin

Streams: Readable

Reading a file **without** streams:

```
1  let http = require('http')
2  let fs = require('fs')
3
4  let server = http.createServer(function (req, res) {
5      fs.readFile(__filename, function (err, data) {
6          res.end(data)
7      })
8  })
9  server.listen(8000)
```

Streams: Readable

Reading a file **without** streams:

Pros:

- Simple

Cons:

- Verbose
- Must store a copy of __*filename* in memory for **every request**
- Must wait for __*filename* to finish loading before sending response

Streams: Readable

Reading a file **with** streams:

```
1  let http = require('http')
2  let fs = require('fs')
3
4  let server = http.createServer(function (req, res) {
5      fs.createReadStream(__filename).pipe(res)
6  })
7  server.listen(8000)
```

Streams: Readable

Reading a file **with** streams:

Pros:

- Simpler code
- Concise
- Low latency
- Low memory pressure

Cons:

- More conceptually complex

Streams: Readable

Add compression. Just 1 line!

```
1  let http = require('http')
2  let fs = require('fs')
3  let oppressor = require('oppressor')
4
5  let server = http.createServer(function (req, res) {
6      fs.createReadStream(__filename)
7        .pipe(oppressor(req))
8        .pipe(res)
9  })
```

Streams: Readable

Readable streams output data that can be inputted into other stream types:

- writable
- transform
- duplex

```
1  readableStream.pipe(writableOrTransformOrDuplexStream)
```

Streams: Readable

Create an arbitrary readable stream:

```
1  let Readable = require('stream').Readable
2
3  let readableStream = new Readable
4  readableStream.push('hello ')
5  readableStream.push('world\n')
6  // Send null to end stream
7  readableStream.push(null)
8
9  readableStream.pipe(process.stdout)
```

Streams: Readable

Implement ._*read()* to build a custom readable stream:

```
1  let Readable = require('stream').Readable
2  let readableStream = Readable()
3
4  let c = 97
5  readableStream._read = () => {
6    readableStream.push(String.fromCharCode(c++))
7    if (c > 'z'.charCodeAt(0)) readableStream.push(null)
8  }
9
10 readableStream.pipe(process.stdout)
```

Streams: Readable

Notes:

- Data can be pushed before or after being piped
  - Similar to promises holding their value
  - Different from promises because they can only be read once
- Readable streams start in **non-flowing** or paused mode
- *.pipe()* unpauses or "resumes" a stream
- Adding a *'data'* event listener will also resume a readable stream

Streams: Readable

Manually consume a readable stream by calling *.read()*:

```
1 process.stdin.on('readable', () => {
2   let buffer = process.stdin.read()
3   console.log(buffer)
4 })
```

Streams: Writable

A writable stream is a stream that **inputs** data.

Examples of writable streams:

- http requests, on the client
- http responses, on the server
- fs write streams
- zlib streams
- crypto streams
- tcp sockets
- child process stdin
- process.stdout
- process.stderr

Streams: Writable

Writing a file **without** streams:

```
let http = require('http')
let fs = require('fs')

let server = http.createServer(function (req, res) {
  let buffer
  req.on('data', data => {
    buffer += data
  })
  req.on('end', () => {
    fs.writeFile(__dirname + '/data.txt', buffer, function (err, data) {
      res.end()
    })
  })
})
server.listen(8000)
```

Streams: Writable

Writing a file **without** streams:

Pros:

- ??? (No longer simple)

Cons:

- Verbose
- Must store a copy of *req.body* in memory for **every request**
- Must wait for *request* to complete before writing file

Streams: Writable

Writing a file **with** streams:

```
let http = require('http')
let fs = require('fs')

let server = http.createServer(function (req, res) {
    req.pipe(fs.createWriteStream(__dirname + '/data.txt'))
})
server.listen(8000)
```

Streams: Writable

Writing a file **with** streams:

Pros:

- Simpler code
- Concise
- Low latency
- Low memory pressure

Cons:

- ~~More conceptually complex~~

Streams: Writable

Writable streams accept input from other stream's output:

- readable
- transform
- duplex

```
1  readableOrTransformOrDuplexStream.pipe(writableStream)
```

Streams: Writable

Create a writable stream:

```
1 let fs = require('fs')
2 let writableStream = fs.createWriteStream(__filename)
3
4 writableStream.write('hello ')
5
6 setImmediate(() => writableStream.end('world\n'))
```

Streams: Writable

Implement ._*write()* to build a custom writable stream:

```
1  let Writable = require('stream').Writable
2  let writableStream = new Writable
3
4  // Implement the _write function to consume data
5  writableStream._write = (chunk, enc, next) => {
6    console.log(chunk)
7    next()
8  }
9
10 process.stdin.pipe(writableStream)
```

Streams: Writable

Notes:

- Data can be pushed before or after being piped
  - Similar to promises holding their value
  - Different from promises because they can only be read once
- Readable streams start in **non-flowing** or paused mode
- *.pipe()* unpauses or "resumes" a stream
- Adding a *'data'* event listener will also resume a readable stream

Streams: Writable

Manually consume a readable stream by calling *.read()*:

```
1 process.stdin.on('readable', () => {
2   let buffer = process.stdin.read()
3   console.log(buffer)
4 })
```

Streams: Transform

A transform (aka “through”) stream both **inputs** and **outputs** data.

A transform stream typically transforms data in some way.

```
1 let through = require('through')
2 process.stdin.pipe(through(console.log, ()=>console.log('\n')))
```

Transform stream examples: zlib streams, crypto streams

Streams: Duplex

A duplex stream both inputs and outputs data.

A duplex stream is a combination readable and writable stream.

(Implements both ._*read()* and ._*write()*)

```
1 let fs = require('fs')
2 let request = require('request')
3 let duplexStream = request('http://google.com')
4 fs.createReadStream(__filename).pipe(duplexStream).pipe(process.stdout)
```

Duplex stream examples: tcp sockets, zlib streams, crypto streams

Streams: Classic or Legacy

- Classic streams are *EventEmitter*s with no buffering
  - aka “push streams”
- Similar API
- Always in “flowing” mode
  - Start out unpaused, which means you can drop ‘data’ events
- Adding a ‘data’ event listener to a stream2 will cause it to fall back into legacy mode
- No special ._*read()* or _.*write()* support
- Must handle buffering and back-pressure manually