# Understanding Streams

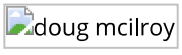
Suggest Edits (/docs/understanding-streams/edit)

(Taken from substack's Stream Handbook)

---

```
"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 (http://cm.bell-labs.com/who/dmr/mdmpipe.html)



Streams come to us from the earliest days of unix (http://www.youtube.com/watch?v=tc4ROCJYbm0) and have proven themselves over the decades as a dependable way to compose large systems out of small components that do one thing well (http://www.faqs.org/docs/artu/ch01s06.html).

In unix, streams are implemented by the shell with `|` pipes. In node, the built-in stream module (http://nodejs.org/docs/latest/api/stream.html) is used by the core libraries and can also be used by user-space modules. Similar to unix, the node stream module's primary composition operator is called `.pipe()` and you get a backpressure mechanism for free to throttle writes for slow consumers.

Streams can help to separate your concerns (http://www.c2.com/cgi/wiki?SeparationOfConcerns) because they restrict the implementation surface area into a consistent interface that can be reused (http://www.faqs.org/docs/artu/ch01s06.html#id2877537). You can then plug the output of one stream to the input of another and use libraries (http://npmjs.org) that operate abstractly on streams to institute higher-level flow control.

Streams are an important component of small-program design (https://michaelochurch.wordpress.com/2012/08/15/what-is-spaghetti-code/) and unix philosophy (http://www.faqs.org/docs/artu/ch01s06.html) but there are many other important abstractions worth considering. Just remember that technical debt (http://c2.com/cgi/wiki?TechnicalDebt) is the enemy and to seek the best abstractions for the problem at hand.


---

# why you should use streams

I/O in node is asynchronous, so interacting with the disk and network involves passing callbacks to functions. You might be tempted to write code that serves up a file from disk like this:

JavaScript ()

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
    fs.readFile(__dirname + '/data.txt', function (err, data) {
        res.end(data);
    });
});
server.listen(8000);
```

This code works but it's bulky and buffers up the entire `data.txt` file into memory for every request before writing the result back to clients. If `data.txt` is very large, your program could start eating a lot of memory as it serves lots of users concurrently, particularly for users on slow connections.

The user experience is poor too because users will need to wait for the whole file to be buffered into memory on your server before they can start receiving any contents.

Luckily both of the `(req, res)` arguments are streams, which means we can write this in a much better way using `fs.createReadStream()` instead of `fs.readFile()`:

```
var http = require('http');
var fs = require('fs');

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

Here `.pipe()` takes care of listening for `'data'` and `'end'` events from the `fs.createReadStream()`. This code is not only cleaner, but now the `data.txt` file will be written to clients one chunk at a time immediately as they are received from the disk.

Using `.pipe()` has other benefits too, like handling backpressure automatically so that node won't buffer chunks into memory needlessly when the remote client is on a really slow or high-latency connection.

Want compression? There are streaming modules for that too!

```
var http = require('http');
var fs = require('fs');
var oppressor = require('oppressor');

var server = http.createServer(function (req, res) {
    var stream = fs.createReadStream(__dirname + '/data.txt');
    stream.pipe(oppressor(req)).pipe(res);
});
server.listen(8000);
```

Now our file is compressed for browsers that support gzip or deflate! We can just let oppressor (https://github.com/substack/oppressor) handle all that content-encoding stuff.

Once you learn the stream api, you can just snap together these streaming modules like lego bricks or garden hoses instead of having to remember how to push data through wonky non-streaming custom APIs.

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

# basics

There are 5 kinds of streams: readable, writable, transform, duplex, and "classic".

### PIPE

All the different types of streams use `.pipe()` to pair inputs with outputs.

`.pipe()` is just a function that takes a readable source stream `src` and hooks the output to a destination writable stream `dst`:

```
src.pipe(dst)
```

`.pipe(dst)` returns `dst` so that you can chain together multiple `.pipe()` calls together:

```
a.pipe(b).pipe(c).pipe(d)
```

which is the same as:

```
a.pipe(b);
b.pipe(c);
c.pipe(d);
```

This is very much like what you might do on the command-line to pipe programs together:

```
a | b | c | d
```

except in node instead of the shell!

## READABLE STREAMS

Readable streams produce data that can be fed into a writable, transform, or duplex stream by calling `.pipe()`:

```
readableStream.pipe(dst)
```

### *creating a readable stream*

Let's make a readable stream!

```
var Readable = require('stream').Readable;

var rs = new Readable;
rs.push('beep ');
rs.push('boop\n');
rs.push(null);

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

```
$ node read0.js
beep boop
```

`rs.push(null)` tells the consumer that `rs` is done outputting data.

Note here that we pushed content to the readable stream `rs` before piping to `process.stdout`, but the complete message was still written.

This is because when you `.push()` to a readable stream, the chunks you push are buffered until a consumer is ready to read them.

However, it would be even better in many circumstances if we could avoid buffering data altogether and only generate the data when the consumer asks for it.

We can push chunks on-demand by defining a `._read` function:

```
var Readable = require('stream').Readable;
var rs = Readable();

var c = 97;
rs._read = function () {
    rs.push(String.fromCharCode(c++));
    if (c > 'z'.charCodeAt(0)) rs.push(null);
};

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

```
$ node read1.js
abcdefghijklmnopqrstuvwxyz
```

Here we push the letters `'a'` through `'z'`, inclusive, but only when the consumer is ready to read them.

The `_read` function will also get a provisional `size` parameter as its first argument that specifies how many bytes the consumer wants to read, but your readable stream can ignore the `size` if it wants.

Note that you can also use `util.inherits()` to subclass a Readable stream, but that approach doesn't lend itself very well to comprehensible examples.

To show that our `_read` function is only being called when the consumer requests, we can modify our readable stream code slightly to add a delay:

```
var Readable = require('stream').Readable;
var rs = Readable();

var c = 97 - 1;

rs._read = function () {
    if (c >= 'z'.charCodeAt(0)) return rs.push(null);

    setTimeout(function () {
        rs.push(String.fromCharCode(++c));
    }, 100);
};

rs.pipe(process.stdout);

process.on('exit', function () {
    console.error('\n_read() called ' + (c - 97) + ' times');
});
process.stdout.on('error', process.exit);
```

Running this program we can see that `_read()` is only called 5 times when we only request 5 bytes of output:

```
$ node read2.js | head -c5
abcde
_read() called 5 times
```

The setTimeout delay is necessary because the operating system requires some time to send us the relevant signals to close the pipe.

The `process.stdout.on('error', fn)` handler is also necessary because the operating system will send a SIGPIPE to our process when `head` is no longer interested in our program's output, which gets emitted as an EPIPE error on `process.stdout`.

These extra complications are necessary when interfacing with the external operating system pipes but are automatic when we interface directly with node streams the whole time.

If you want to create a readable stream that pushes arbitrary values instead of just strings and buffers, make sure to create your readable stream with `Readable({ objectMode: true })`.

### *consuming a readable stream*

Most of the time it's much easier to just pipe a readable stream into another kind of stream or a stream created with a module like through (https://npmjs.org/package/through) or concat-stream (https://npmjs.org/package/concat-stream), but occasionally it might be useful to consume a readable stream directly.

```
process.stdin.on('readable', function () {
    var buf = process.stdin.read();
    console.dir(buf);
});
```

```
$ (echo abc; sleep 1; echo def; sleep 1; echo ghi) | node consume0.js
<Buffer 61 62 63 0a>
<Buffer 64 65 66 0a>
<Buffer 67 68 69 0a>
null
```

When data is available, the `'readable'` event fires and you can call `.read()` to fetch some data from the buffer.

When the stream is finished, `.read()` returns `null` because there are no more bytes to fetch.

You can also tell `.read(n)` to return `n` bytes of data. Reading a number of bytes is merely advisory and does not work for object streams, but all of the core streams support it.

Here's an example of using `.read(n)` to buffer stdin into 3-byte chunks:

```
process.stdin.on('readable', function () {
    var buf = process.stdin.read(3);
    console.dir(buf);
});
```

Running this example gives us incomplete data!

```
$ (echo abc; sleep 1; echo def; sleep 1; echo ghi) | node consume1.js
<Buffer 61 62 63>
<Buffer 0a 64 65>
<Buffer 66 0a 67>
```

This is because there is extra data left in internal buffers and we need to give node a "kick" to tell it that we are interested in more data past the 3 bytes that we've already read. A simple `.read(0)` will do this:

```
process.stdin.on('readable', function () {
    var buf = process.stdin.read(3);
    console.dir(buf);
    process.stdin.read(0);
});
```

Now our code works as expected in 3-byte chunks!

```
$ (echo abc; sleep 1; echo def; sleep 1; echo ghi) | node consume2.js
<Buffer 61 62 63>
<Buffer 0a 64 65>
<Buffer 66 0a 67>
<Buffer 68 69 0a>
```

You can also use `.unshift()` to put back data so that the same read logic will fire when `.read()` gives you more data than you wanted.

Using `.unshift()` prevents us from making unnecessary buffer copies. Here we can build a readable parser to split on newlines:

```
var offset = 0;

process.stdin.on('readable', function () {
    var buf = process.stdin.read();
    if (!buf) return;
    for (; offset < buf.length; offset++) {
        if (buf[offset] === 0x0a) {
            console.dir(buf.slice(0, offset).toString());
            buf = buf.slice(offset + 1);
            offset = 0;
            process.stdin.unshift(buf);
            return;
        }
    }
    process.stdin.unshift(buf);
});
```

```
$ tail -n +50000 /usr/share/dict/american-english | head -n10 | node lines.js
'hearties'
'heartiest'
'heartily'
'heartiness'
'heartiness\'s'
'heartland'
'heartland\'s'
'heartlands'
'heartless'
'heartlessly'
```

However, there are modules on npm such as split (https://npmjs.org/package/split) that you should use instead of rolling your own line-parsing logic.

## WRITABLE STREAMS

A writable stream is a stream you can `.pipe()` to but not from:

```
src.pipe(writableStream)
```

### *creating a writable stream*

Just define a `._write(chunk, enc, next)` function and then you can pipe a readable stream in:

```
var Writable = require('stream').Writable;
var ws = Writable();
ws._write = function (chunk, enc, next) {
    console.dir(chunk);
    next();
};

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

```
$ (echo beep; sleep 1; echo boop) | node write0.js
<Buffer 62 65 65 70 0a>
<Buffer 62 6f 6f 70 0a>
```

The first argument, `chunk` is the data that is written by the producer.

The second argument `enc` is a string with the string encoding, but only when `opts.decodeString` is `false` and you've been written a string.

The third argument, `next(err)` is the callback that tells the consumer that they can write more data. You can optionally pass an error object `err`, which emits an `'error'` event on the stream instance.

If the readable stream you're piping from writes strings, they will be converted into `Buffer`s unless you create your writable stream with `Writable({ decodeStrings: false })`.

If the readable stream you're piping from writes objects, create your writable stream with `Writable({ objectMode: true })`.

### *writing to a writable stream*

To write to a writable stream, just call `.write(data)` with the `data` you want to write!

```
process.stdout.write('beep boop\n');
```

To tell the destination writable stream that you're done writing, just call `.end()`. You can also give `.end(data)` some `data` to write before ending:

```
var fs = require('fs');
var ws = fs.createWriteStream('message.txt');

ws.write('beep ');

setTimeout(function () {
    ws.end('boop\n');
}, 1000);
```

```
$ node writing1.js
$ cat message.txt
beep boop
```

If you care about high water marks and buffering, `.write()` returns false when there is more data than the `opts.highWaterMark` option passed to `Writable()` in the incoming buffer.

If you want to wait for the buffer to empty again, listen for a `'drain'` event.

## TRANSFORM

Transform streams are a certain type of duplex stream (both readable and writable). The distinction is that in Transform streams, the output is in some way calculated from the input.

You might also hear transform streams referred to as "through streams".

Through streams are simple readable/writable filters that transform input and produce output.

## DUPLEX

Duplex streams are readable/writable and both ends of the stream engage in a two-way interaction, sending back and forth messages like a telephone. An rpc exchange is a good example of a duplex stream. Any time you see something like:

```
a.pipe(b).pipe(a)
```

you're probably dealing with a duplex stream.

## CLASSIC STREAMS

Classic streams are the old interface that first appeared in node 0.4.
You will probably encounter this style of stream for a long time so it's good to know how they work.

Whenever a stream has a `"data"` listener registered, it switches into `"classic"` mode and behaves according to the old API.

### *classic readable streams*

Classic readable streams are just event emitters that emit `"data"` events when they have data for their consumers and emit `"end"` events when they are done producing data for their consumers.

`.pipe()` checks whether a classic stream is readable by checking the truthiness of `stream.readable`.

Here is a super simple readable stream that prints `A` through `J`, inclusive:

```
var Stream = require('stream');
var stream = new Stream;
stream.readable = true;

var c = 64;
var iv = setInterval(function () {
    if (++c >= 75) {
        clearInterval(iv);
        stream.emit('end');
    }
    else stream.emit('data', String.fromCharCode(c));
}, 100);

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

```
$ node classic0.js
ABCDEFGHIJ
```

To read from a classic readable stream, you register `"data"` and `"end"` listeners. Here's an example reading from `process.stdin` using the old readable stream style:

```
process.stdin.on('data', function (buf) {
    console.log(buf);
});
process.stdin.on('end', function () {
    console.log('__END__');
});
```

```
$ (echo beep; sleep 1; echo boop) | node classic1.js
<Buffer 62 65 65 70 0a>
<Buffer 62 6f 6f 70 0a>
__END__
```

Note that whenever you register a `"data"` listener, you put the stream into compatability mode so you lose the benefits of the new streams2 api.

You should pretty much never register `"data"` and `"end"` handlers yourself anymore. If you need to interact with legacy streams, use libraries that you can `.pipe()` to instead where possible.

For example, you can use through (https://npmjs.org/package/through) to avoid setting up explicit `"data"` and `"end"` listeners:

```
var through = require('through');
process.stdin.pipe(through(write, end));

function write (buf) {
    console.log(buf);
}
function end () {
    console.log('__END__');
}
```

```
$ (echo beep; sleep 1; echo boop) | node through.js
<Buffer 62 65 65 70 0a>
<Buffer 62 6f 6f 70 0a>
__END__
```

or use concat-stream (https://npmjs.org/package/concat-stream) to buffer up an entire stream's contents:

```
var concat = require('concat-stream');
process.stdin.pipe(concat(function (body) {
    console.log(JSON.parse(body));
}));
```

```
$ echo '{"beep":"boop"}' | node concat.js
{ beep: 'boop' }
```

Classic readable streams have `.pause()` and `.resume()` logic for provisionally pausing a stream, but this was merely advisory. If you are going to use `.pause()` and `.resume()` with classic readable streams, you should use through (https://npmjs.org/package/through) to handle buffering instead of writing that yourself.

### *classic writable streams*

Classic writable streams are very simple. Just define `.write(buf)`, `.end(buf)` and `.destroy()`.

`.end(buf)` may or may not get a `buf`, but node people will expect `stream.end(buf)` to mean `stream.write(buf); stream.end()` and you shouldn't violate their expectations.

### READ MORE

- core stream documentation (http://nodejs.org/docs/latest/api/stream.html#stream_stream)
- You can use the readable-stream (https://npmjs.org/package/readable-stream) module to make your streams2 code compliant with node 0.8 and below. Just `require('readable-stream')` instead of `require('stream')` after you `npm install readable-stream`.

---

## built-in streams

These streams are built into node itself.

### PROCESS

*process.stdin (http://nodejs.org/docs/latest/api/process.html#process_process_stdin)*

This readable stream contains the standard system input stream for your program.

It is paused by default but the first time you refer to it `.resume()` will be called implicitly on the next tick (http://nodejs.org/docs/latest/api/process.html#process_process_nexttick_callback).

If process.stdin is a tty (check with `tty.isatty()` (http://nodejs.org/docs/latest/api/tty.html#tty_tty_isatty_fd)) then input events will be line-buffered. You can turn off line-buffering by calling `process.stdin.setRawMode(true)` BUT the default handlers for key combinations such as `^C` and `^D` will be removed.

*process.stdout (http://nodejs.org/api/process.html#process_process_stdout)*

This writable stream contains the standard system output stream for your program.

`write` to it if you want to send data to stdout

*process.stderr (http://nodejs.org/api/process.html#process_process_stderr)*

This writable stream contains the standard system error stream for your program.

`write` to it if you want to send data to stderr

### CHILD_PROCESS.SPAWN()

### FS

*fs.createReadStream()*

*fs.createWriteStream()*

### NET

*net.connect() (http://nodejs.org/docs/latest/api/net.html#net_net_connect_options_connectionlistener)*

This function returns a [duplex stream] that connects over tcp to a remote
host.

You can start writing to the stream right away and the writes will be buffered
until the `'connect'` event fires.

*net.createServer()*

### HTTP

*http.request()*

*http.createServer()*

### ZLIB

*zlib.createGzip()*

*zlib.createGunzip()*

*zlib.createDeflate()*

*zlib.createInflate()*

---

# control streams

### THROUGH (HTTPS://GITHUB.COM/DOMINICTARR/THROUGH)

### FROM (HTTPS://GITHUB.COM/DOMINICTARR/FROM)

### PAUSE-STREAM (HTTPS://GITHUB.COM/DOMINICTARR/PAUSE-STREAM)

### CONCAT-STREAM (HTTPS://GITHUB.COM/MAXOGDEN/NODE-CONCAT-STREAM)

concat-stream will buffer up stream contents into a single buffer.
`concat(cb)` just takes a single callback `cb(body)` with the buffered
`body` when the stream has finished.

For example, in this program, the concat callback fires with the body string
`"beep boop"` once `cs.end()` is called.
The program takes the body and upper-cases it, printing `BEEP BOOP`.

```
var concat = require('concat-stream');

var cs = concat(function (body) {
    console.log(body.toUpperCase());
});
cs.write('beep ');
cs.write('boop.');
cs.end();
```

```
$ node concat.js
BEEP BOOP.
```

Here's an example usage of concat-stream that will parse incoming url-encoded
form data and reply with a stringified JSON version of the form parameters:

```
var http = require('http');
var qs = require('querystring');
var concat = require('concat-stream');

var server = http.createServer(function (req, res) {
    req.pipe(concat(function (body) {
        var params = qs.parse(body.toString());
        res.end(JSON.stringify(params) + '\n');
    }));
});
server.listen(5005);
```

```
$ curl -X POST -d 'beep=boop&dinosaur=trex' http://localhost:5005
{"beep":"boop","dinosaur":"trex"}
```

### DUPLEX (HTTPS://GITHUB.COM/DOMINICTARR/DUPLEX)

### DUPLEXER (HTTPS://GITHUB.COM/RAYNOS/DUPLEXER)

### EMIT-STREAM (HTTPS://GITHUB.COM/SUBSTACK/EMIT-STREAM)

### INVERT-STREAM (HTTPS://GITHUB.COM/DOMINICTARR/INVERT-STREAM)

### MAP-STREAM (HTTPS://GITHUB.COM/DOMINICTARR/MAP-STREAM)

### REMOTE-EVENTS (HTTPS://GITHUB.COM/DOMINICTARR/REMOTE-EVENTS)

### BUFFER-STREAM (HTTPS://GITHUB.COM/RAYNOS/BUFFER-STREAM)

### EVENT-STREAM (HTTPS://GITHUB.COM/DOMINICTARR/EVENT-STREAM)

### AUTH-STREAM (HTTPS://GITHUB.COM/RAYNOS/AUTH-STREAM)

---

# meta streams

### MUX-DEMUX (HTTPS://GITHUB.COM/DOMINICTARR/MUX-DEMUX)

### STREAM-ROUTER (HTTPS://GITHUB.COM/RAYNOS/STREAM-ROUTER)

### MULTI-CHANNEL-MDM (HTTPS://GITHUB.COM/RAYNOS/MULTI-CHANNEL-MDM)

---

# state streams

### CRDT (HTTPS://GITHUB.COM/DOMINICTARR/CRDT)

### DELTA-STREAM (HTTPS://GITHUB.COM/RAYNOS/DELTA-STREAM)

### SCUTTLEBUTT (HTTPS://GITHUB.COM/DOMINICTARR/SCUTTLEBUTT)

scuttlebutt (https://github.com/dominictarr/scuttlebutt) can be used for peer-to-peer state synchronization with a mesh topology where nodes might only be connected through intermediaries and there is no node with an authoritative version of all the data.

The kind of distributed peer-to-peer network that scuttlebutt (https://github.com/dominictarr/scuttlebutt) provides is especially useful when nodes on different sides of network barriers need to share and update the same state. An example of this kind of network might be browser clients that send messages through an http server to each other and backend processes that the browsers can't directly connect to. Another use-case might be systems that span internal networks since IPv4 addresses are scarce.

scuttlebutt (https://github.com/dominictarr/scuttlebutt) uses a gossip protocol (https://en.wikipedia.org/wiki/Gossip_protocol) to pass messages between connected nodes so that state across all the nodes will eventually converge (https://en.wikipedia.org/wiki/Eventual_consistency) on the same value everywhere.

Using the `scuttlebutt/model` interface, we can create some nodes and pipe them to each other to create whichever sort of network we want:

```
var Model = require('scuttlebutt/model');
var am = new Model;
var as = am.createStream();

var bm = new Model;
var bs = bm.createStream();

var cm = new Model;
var cs = cm.createStream();

var dm = new Model;
var ds = dm.createStream();

var em = new Model;
var es = em.createStream();

as.pipe(bs).pipe(as);
bs.pipe(cs).pipe(bs);
bs.pipe(ds).pipe(bs);
ds.pipe(es).pipe(ds);

em.on('update', function (key, value, source) {
    console.log(key + ' => ' + value + ' from ' + source);
});

am.set('x', 555);
```

The network we've created is an undirected graph that looks like:

```
a <-> b <-> c
      ^
      |
      v
      d <-> e
```

Note that nodes `a` and `e` aren't directly connected, but when we run this script:

```
$ node model.js
x => 555 from 1347857300518
```

the value that node `a` set finds its way to node `e` by way of nodes `b` and `d`. Here all the nodes are in the same process but because scuttlebutt (https://github.com/dominictarr/scuttlebutt) uses a simple streaming interface, the nodes can be placed on any process or server and connected with any streaming transport that can handle string data.

Next we can make a more realistic example that connects over the network and increments a counter variable.

Here's the server which will set the initial `count` value to 0 and `count ++` every 320 milliseconds, printing all updates to count:

```
var Model = require('scuttlebutt/model');
var net = require('net');

var m = new Model;
m.set('count', '0');
m.on('update', function (key, value) {
    console.log(key + ' = ' + m.get('count'));
});

var server = net.createServer(function (stream) {
    stream.pipe(m.createStream()).pipe(stream);
});
server.listen(8888);

setInterval(function () {
    m.set('count', Number(m.get('count')) + 1);
}, 320);
```

Now we can make a client that connects to this server, updates the count on an interval, and prints all the updates it receives:

```
var Model = require('scuttlebutt/model');
var net = require('net');

var m = new Model;
var s = m.createStream();

s.pipe(net.connect(8888, 'localhost')).pipe(s);

m.on('update', function cb (key) {
    // wait until we've gotten at least one count value from the network
    if (key !== 'count') return;
    m.removeListener('update', cb);

    setInterval(function () {
        m.set('count', Number(m.get('count')) + 1);
    }, 100);
});

m.on('update', function (key, value) {
    console.log(key + ' = ' + value);
});
```

The client is slightly trickier since it should wait until it has an update from somebody else to start updating the counter itself or else its counter would be zeroed.

Once we get the server and some clients running we should see a sequence like this:

```
count = 183
count = 184
count = 185
count = 186
count = 187
count = 188
count = 189
```

Occasionally on some of the nodes we might see a sequence with repeated values like:

```
count = 147
count = 148
count = 149
count = 149
count = 150
count = 151
```

These values are due to
scuttlebutt's (https://github.com/dominictarr/scuttlebutt)
history-based conflict resolution algorithm which is hard at work ensuring that the state of the system across all nodes is eventually consistent.

Note that the server in this example is just another node with the same
privledges as the clients connected to it. The terms "client" and "server" here
don't affect how the state synchronization proceeds, just who initiates the
connection. Protocols with this property are often called symmetric protocols.
See dnode (https://github.com/substack/dnode) for another example of a
symmetric protocol.

### APPEND-ONLY (HTTP://GITHUB.COM/RAYNOS/APPEND-ONLY)

---

## http streams

### REQUEST (HTTPS://GITHUB.COM/MIKEAL/REQUEST)

### OPPRESSOR (HTTPS://GITHUB.COM/SUBSTACK/OPPRESSOR)

### RESPONSE-STREAM (HTTPS://GITHUB.COM/SUBSTACK/RESPONSE-STREAM)

---

## io streams

### RECONNECT (HTTPS://GITHUB.COM/DOMINICTARR/RECONNECT)

### KV (HTTPS://GITHUB.COM/DOMINICTARR/KV)

### DISCOVERY-NETWORK (HTTPS://GITHUB.COM/RAYNOS/DISCOVERY-NETWORK)

---

## parser streams

### TAR (HTTPS://GITHUB.COM/CREATIONIX/NODE-TAR)

### TRUMPET (HTTPS://GITHUB.COM/SUBSTACK/NODE-TRUMPET)

### JSONSTREAM (HTTPS://GITHUB.COM/DOMINICTARR/JSONSTREAM)

Use this module to parse and stringify json data from streams.

If you need to pass a large json collection through a slow connection or you have a json object that will populate slowly this module will let you parse data incrementally as it arrives.

### JSON-SCRAPE (HTTPS://GITHUB.COM/SUBSTACK/JSON-SCRAPE)

### STREAM-SERIALIZER (HTTPS://GITHUB.COM/DOMINICTARR/STREAM-SERIALIZER)

---

## browser streams

### SHOE (HTTPS://GITHUB.COM/SUBSTACK/SHOE)

### DOMNODE (HTTPS://GITHUB.COM/MAXOGDEN/DOMNODE)

### SORTA (HTTPS://GITHUB.COM/SUBSTACK/SORTA)

### GRAPH-STREAM (HTTPS://GITHUB.COM/SUBSTACK/GRAPH-STREAM)

### ARROW-KEYS (HTTPS://GITHUB.COM/RAYNOS/ARROW-KEYS)

### ATTRIBUTE (HTTPS://GITHUB.COM/RAYNOS/ATTRIBUTE)

### DATA-BIND (HTTPS://GITHUB.COM/TRAVIS4ALL/DATA-BIND)

---

## html streams

### HYPERSTREAM (HTTPS://GITHUB.COM/SUBSTACK/HYPERSTREAM)

## audio streams

### BAUDIO (HTTPS://GITHUB.COM/SUBSTACK/BAUDIO)

## rpc streams

### DNODE (HTTPS://GITHUB.COM/SUBSTACK/DNODE)

dnode (https://github.com/substack/dnode) lets you call remote functions through any kind of stream.

Here's a basic dnode server:

```
var dnode = require('dnode');
var net = require('net');

var server = net.createServer(function (c) {
    var d = dnode({
        transform : function (s, cb) {
            cb(s.replace(/[aeiou]{2,}/, 'oo').toUpperCase())
        }
    });
    c.pipe(d).pipe(c);
});

server.listen(5004);
```

then you can hack up a simple client that calls the server's `.transform()` function:

```
var dnode = require('dnode');
var net = require('net');

var d = dnode();
d.on('remote', function (remote) {
    remote.transform('beep', function (s) {
        console.log('beep => ' + s);
        d.end();
    });
});

var c = net.connect(5004);
c.pipe(d).pipe(c);
```

Fire up the server, then when you run the client you should see:

```
$ node client.js
beep => BOOP
```

The client sent `'beep'` to the server's `transform()` function and the server called the client's callback with the result, neat!

The streaming interface that dnode provides here is a duplex stream since both the client and server are piped to each other ( `c.pipe(d).pipe(c)` ) with requests and responses coming from both sides.

The craziness of dnode begins when you start to pass function arguments to stubbed callbacks. Here's an updated version of the previous server with a multi-stage callback passing dance:

```
var dnode = require('dnode');
var net = require('net');

var server = net.createServer(function (c) {
    var d = dnode({
        transform : function (s, cb) {
            cb(function (n, fn) {
                var oo = Array(n+1).join('o');
                fn(s.replace(/[aeiou]{2,}/, oo).toUpperCase());
            });
        }
    });
    c.pipe(d).pipe(c);
});

server.listen(5004);
```

Here's the updated client:

```
var dnode = require('dnode');
var net = require('net');

var d = dnode();
d.on('remote', function (remote) {
    remote.transform('beep', function (cb) {
        cb(10, function (s) {
            console.log('beep:10 => ' + s);
            d.end();
        });
    });
});

var c = net.connect(5004);
c.pipe(d).pipe(c);
```

After we spin up the server, when we run the client now we get:

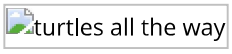
```
$ node client.js
beep:10 => BOOOOOOOOOP
```

It just works!™

The basic idea is that you just put functions in objects and you call them from the other side of a stream and the functions will be stubbed out on the other end to do a round-trip back to the side that had the original function in the first place. The best thing is that when you pass functions to a stubbed function as arguments, those functions get stubbed out on the *other* side!

This approach of stubbing function arguments recursively shall henceforth be known as the "turtles all the way down" gambit. The return values of any of your functions will be ignored and only enumerable properties on objects will be sent, json-style.

It's turtles all the way down!



Since dnode works in node or on the browser over any stream it's easy to call functions defined anywhere and especially useful when paired up with mux-demux (https://github.com/dominictarr/mux-demux) to multiplex an rpc stream for control alongside some bulk data streams.

### RPC-STREAM (HTTPS://GITHUB.COM/DOMINICTARR/RPC-STREAM)

---

# test streams

### TAP (HTTPS://GITHUB.COM/ISAACS/NODE-TAP)

### STREAM-SPEC (HTTPS://GITHUB.COM/DOMINICTARR/STREAM-SPEC)

---

# power combos

### DISTRIBUTED PARTITION-TOLERANT CHAT

The append-only (http://github.com/Raynos/append-only) module can give us a convenient append-only array on top of scuttlebutt (https://github.com/dominictarr/scuttlebutt) which makes it really easy to write an eventually-consistent, distributed chat that can replicate with other nodes and survive network partitions.

TODO: the rest

### ROLL YOUR OWN SOCKET.IO

We can build a socket.io-style event emitter api over streams using some of the libraries mentioned earlier in this document.

First we can use shoe (http://github.com/substack/shoe) to create a new websocket handler server-side and emit-stream (https://github.com/substack/emit-stream) to turn an event emitter into a stream that emits objects. The object stream can then be fed into JSONStream (https://github.com/dominictarr/JSONStream) to serialize the objects and from there the serialized stream can be piped into the remote browser.

```
var EventEmitter = require('events').EventEmitter;
var shoe = require('shoe');
var emitStream = require('emit-stream');
var JSONStream = require('JSONStream');

var sock = shoe(function (stream) {
    var ev = new EventEmitter;
    emitStream(ev)
        .pipe(JSONStream.stringify())
        .pipe(stream)
    ;
    ...
});
```

Inside the shoe callback we can emit events to the `ev` function. Here we'll just emit different kinds of events on intervals:

```
var intervals = [];

intervals.push(setInterval(function () {
    ev.emit('upper', 'abc');
}, 500));

intervals.push(setInterval(function () {
    ev.emit('lower', 'def');
}, 300));

stream.on('end', function () {
    intervals.forEach(clearInterval);
});
```

Finally the shoe instance just needs to be bound to an http server:

```
var http = require('http');
var server = http.createServer(require('ecstatic')(__dirname));
server.listen(8080);

sock.install(server, '/sock');
```

Meanwhile on the browser side of things just parse the json shoe stream and pass the resulting object stream to `eventStream()`. `eventStream()` just returns an event emitter that emits the server-side events:

```
var shoe = require('shoe');
var emitStream = require('emit-stream');
var JSONStream = require('JSONStream');

var parser = JSONStream.parse([true]);
var stream = parser.pipe(shoe('/sock')).pipe(parser);
var ev = emitStream(stream);

ev.on('lower', function (msg) {
    var div = document.createElement('div');
    div.textContent = msg.toLowerCase();
    document.body.appendChild(div);
});

ev.on('upper', function (msg) {
    var div = document.createElement('div');
    div.textContent = msg.toUpperCase();
    document.body.appendChild(div);
});
```

Use browserify (https://github.com/substack/node-browserify) to build this browser source code so that you can `require()` all these nifty modules browser-side:

```
$ browserify main.js -o bundle.js
```

Then drop a `<script src="/bundle.js"></script>` into some html and open it up in a browser to see server-side events streamed through to the browser side of things.

With this streaming approach you can rely more on tiny reusable components that only need to know how to talk streams. Instead of routing messages through a global event system socket.io-style, you can focus more on breaking up your application into tinier units of functionality that can do exactly one thing well.

For instance you can trivially swap out JSONStream in this example for stream-serializer (https://github.com/dominictarr/stream-serializer) to get a different take on serialization with a different set of tradeoffs.
You could bolt layers over top of shoe to handle reconnections (https://github.com/dominictarr/reconnect) or heartbeats using simple streaming interfaces.
You could even add a stream into the chain to use namespaced events with eventemitter2 (https://npmjs.org/package/eventemitter2) instead of the EventEmitter in core.

If you want some different streams that act in different ways it would likewise be pretty simple to run the shoe stream in this example through mux-demux to create separate channels for each different kind of stream that you need.

As the requirements of your system evolve over time, you can swap out each of these streaming pieces as necessary without as many of the all-or-nothing risks that more opinionated framework approaches necessarily entail.

## HTML STREAMS FOR THE BROWSER AND THE SERVER

We can use some streaming modules to reuse the same html rendering logic for the client and the server! This approach is indexable, SEO-friendly, and gives us realtime updates.

Our renderer takes lines of json as input and returns html strings as its output. Text, the universal interface!

render.js:

```
var through = require('through');
var hyperglue = require('hyperglue');
var fs = require('fs');
var html = fs.readFileSync(__dirname + '/static/row.html', 'utf8');

module.exports = function () {
    return through(function (line) {
        try { var row = JSON.parse(line) }
        catch (err) { return this.emit('error', err) }

        this.queue(hyperglue(html, {
            '.who': row.who,
            '.message': row.message
        }).outerHTML);
    });
};
```

We can use brfs (http://github.com/substack/brfs) to inline the `fs.readFileSync()` call for browser code and hyperglue (https://github.com/substack/hyperglue) to update html based on css selectors. You don't need to use hyperglue necessarily here; anything that can return a string with html in it will work.

The `row.html` used is just a really simple stub thing:

row.html:

```
<div class="row">
  <div class="who"></div>
  <div class="message"></div>
</div>
```

The server will just use slice-file (https://github.com/substack/slice-file) to keep everything simple. slice-file (https://github.com/substack/slice-file) is little more than a glorified `tail/tail -f` api but the interfaces map well to databases with regular results plus a changes feed like couchdb.

server.js:

```
var http = require('http');
var fs = require('fs');
var hyperstream = require('hyperstream');
var ecstatic = require('ecstatic')(__dirname + '/static');

var sliceFile = require('slice-file');
var sf = sliceFile(__dirname + '/data.txt');

var render = require('./render');

var server = http.createServer(function (req, res) {
    if (req.url === '/') {
        var hs = hyperstream({
            '#rows': sf.slice(-5).pipe(render())
        });
        hs.pipe(res);
        fs.createReadStream(__dirname + '/static/index.html').pipe(hs);
    }
    else ecstatic(req, res)
});
server.listen(8000);

var shoe = require('shoe');
var sock = shoe(function (stream) {
    sf.follow(-1,0).pipe(stream);
});
sock.install(server, '/sock');
```

The first part of the server handles the `/` route and streams the last 5 lines from `data.txt` into the `#rows` div.

The second part of the server handles realtime updates to `#rows` using shoe (http://github.com/substack/shoe), a simple streaming websocket polyfill.

Next we can write some simple browser code to populate the realtime updates from shoe (http://github.com/substack/shoe) into the `#rows` div:

```
var through = require('through');
var render = require('./render');

var shoe = require('shoe');
var stream = shoe('/sock');

var rows = document.querySelector('#rows');
stream.pipe(render()).pipe(through(function (html) {
    rows.innerHTML += html;
}));
```

Just compile with browserify (http://browserify.org) and brfs (http://github.com/substack/brfs):

```
$ browserify -t brfs browser.js > static/bundle.js
```

And that's it! Now we can populate `data.txt` with some silly data: