3 Sessions

Streams

Node Training

Streams

- Overview
- Code Walkthrough
- Exercises

Introducing Streams

Streams are like Arrays, but laid out in time, rather than in memory
– Jed Shmidt

connecting programs like garden hose--screw
– Doug McIlroy

Introducing Streams

- Derived from the Unix Philosophy
  - Same idea as Unix pipes
  - Particular majors on
    - Simple parts (Rule of Modularity)
    - Connectable parts with a standard interface (Rule of Composition)
    - Data holds the knowledge whilst logic is simple and robust (Rule of Representation)

Introducing Streams

- Core concept in Node
- The paragon of data flow handling
- Handle lots of data in small pieces
- Enabled by Node's event-driven IO Model
  - Asynchronous IO makes it easy to handle data chunk by chunk
- Provides a programming model that is easy to reason about
  - Simply follow the pipe line

Buffering Data

Prone to falling over, delayed results

```
var request = require('request');

request('http://registry.npmjs.org/-/all',
  function (err, res) {
     var recs = JSON.parse(res.body);
    var names = '';

    Object.keys(recs).forEach(function (key) {
     names += recs[key].name + '\n';
    });

    console.log(names);

});
```

nearForm

Streaming Data

Immediate results, low memory usage

```
var request = require('request');
var JSONStream = require('JSONStream');

request('http://registry.npmjs.org/-/all')
  .pipe(JSONStream.parse('*.name', function (name) {
      return name + '\n';
  }))
  .pipe(process.stdout)
```

Streaming Data

Immediate results, low memory usage

```
var request = require('request');
var JSONStream = require('JSONStream');

request('http://registry.npmjs.org/-/all')
  .pipe(JSONStream.parse('*.name', function (name) {
      return name + '\n';
  }))
  .pipe(process.stdout)
```

Stream Types

- **Readable**
  - get data (in chunks) from an input source
  - Example: Standard Input
- **Writable**
  - send data (in chunks) to an output sink
  - Example: Standard Output
- **Duplex**
  - both Readable and Writable
  - Example: network sockets

Stream Types

- **Transform**
  - implements the Duplex stream
  - change the data chunks as they pass through
  - Example: compression
- **PassThrough**
  - Implements the Transform stream, but changes nothing
  - Can be used to hook up non-stream events into a streaming architecture
  - Example: logging

Connecting Streams

```
readable.pipe(writable)
```

- Streams are connected with the `pipe` method
- Data flow is from left to right
- The pipe method exists on streams with read capabilities
- The pipe method takes a single argument
  - A stream with a writable interface

Connecting Streams Walkthrough

```
$ node pipe0.js
```

- Outputs its own file content to standard out.
  - Loads the filesystem module `fs`
  - Uses `fs.createReadStream` to make a readable stream
  - Assigns created stream to the `inStream` variable
  - Pipes inStream into `process.stdout`
    - `process.stdout` is a preexisting writable stream that comes with the Node environment

Event Emitters

Implementing an EventEmitter

```
var EventEmitter = require('events').EventEmitter;
var util = require('util');

function Dog() {
  setInterval(function (self) {
    self.emit('bark', 'woof');
  }, 1000, this);
}
util.inherits(Dog, EventEmitter);

var rufus = new Dog;
rufus.on('bark', function (sound) {
  console.log('%s - shh rufus!', sound);
});
```

Stream Events

- Streams are an EventEmitter implementation
- Errors, readiness, content and stream state is all communicated via emitted events
- Pipes should always be preferred over event listeners

Stream Events

- There are some occasions where listening for an event is necessary
  - An `error` event
  - A `drain` event when manually handling back pressure
  - A `finish` or `end` event where other non-stream abstractions are waiting for all data to be processed

Stream Events Walkthrough

```
$ node request0.js
```

- Retrieves a file over HTTP, streams it into a file, handles errors, notifies when done
- Uses core `fs` and third party `request` modules
- Call to request creates a stream from example.com/file.txt

Stream Events Walkthrough

```
$ node request0.js
```

- Register error listener to catch any connection issues
- `pipe` into write stream, writing to file.txt
- Register `finish` listener that logs when write stream is done
- Attach an error listener to catch any IO problems

Readable Stream Events

- **readable**: there's data waiting
- **data**: a chunk of data
- **end**: no more data!
- **close**: underlying resource has closed
- **error**: there was an error getting data

Writable Stream Events

- **pipe**: a readable stream has been attached
- **unpipe**: a readable stream has been detached
- **finish**: no more data!
- **drain**: it's ok to start sending data again
- **error**: there was an error getting data

Pipelines

```
readable
  .pipe(transform)
  .pipe(writable)
```

- The return value of `pipe` is the stream passed into it
- A pipe can be attached to a returned output stream, when that stream can also accept input (e.g. when a readwrite stream was passed to the previous pipe invocation)
- JavaScript syntax allows methods to be called on new lines, so in this case data flow is top down

Pipelines Walkthrough

```
$ node zip0.js
```

- Creates a compressed version of itself
- Loads core `fs` and `zlib` modules
- Makes a gzip transform stream with `zlib.createGzip`, assign this transform stream to the gzip variable
- Assigns a read stream from `zip0.js` to `inStream`

Pipelines Walkthrough

```
$ node zip0.js
```

- Assigns a write stream into `zip0.js.gz` to `outStream`
- Pipes `inStream` through gzip transform stream and into outStream
  - Data from `zip0.js` passes through compression stream and out into `zip0.js.gz` file

Creating Streams

- The core stream module supplies constructors that be used to instantiate custom streams
  - Directly with the `new` keyword
  - Or through prototypical inheritance

Creating Streams

- Stream implementations requires an assigned callback that's fired upon each read/write chunk
  - Not dissimilar from the early browser API's, e.g. `domElement.onclick = function(){}`
  - Except with streams it makes sense to limit the interface to one callback

Creating Readable Streams

```
var R = require('stream').Readable;
var readable = new R;
readable._read = function (size) {
  this.push(':-)');
  Math.random() > .8 && this.push(null);
}
```

- The `_read` method is required, if it's not implemented the stream will throw
- The `size` parameter is advisory
- Call `this.push` to supply read data
- Pass `null` to push to indicate end of stream

Creating Writable Streams

```
var W = require('stream').Writable;
var writable = new W;
writable._write = function (data, enc, done) {
  //do something data
  done();
}
```

- The `_write` method is required
- The `enc` parameter isn't always necessary
- Call `done` after each `data` chunk has been processed, allows for asynchronous activities.

Back Pressure

- What happens when there is too much data?
  - Think of an overflowing funnel
- Example:
  - Readable stream is fast
  - Writable stream is slow
  - Writable stream should pause the Readable stream
- Back pressure is handled automatically by all streaming instances supplied by core
  - Just pipe streams together and back pressure comes free

Back Pressure

- When creating a custom stream, back pressure may need to be considered
  - Call `pause` on an incoming stream to apply back pressure
  - Call `resume` when ready to receive more data
- Most user created streams are transform streams
  - Core readwrite stream constructs internally mediate back pressure between readable and writeables
- Manually handling back pressure tends to only be required when creating streams that interact with something outside of the normal eco system
  - (e.g a third party C library)

Stream Options

- Stream classes accept an options object
- `objectMode`
  - for object streaming
- `highWaterMark`
  - how much data to buffer before applying back pressure
- `decodeStrings`
  - write streams only, convert incoming strings into binary format (using the core Buffer constructor)
- `encoding`
  - read streams only, convert incoming binary data into a string using the encoding specified
    - utf8, ascii, hex, base64, utf16le

Creating Duplex Streams

- User implementation of both the `_read` and `_write` methods is required
- There doesn't need to be any causal relationship between input and output
- The idea of `Duplex` streams is they can be piped to themselves
  - an echo server
  - the beginning and end of a transform pipeline
- Duplex streams are the basis of `Transform` streams

Creating Transform Streams

- Most common type of user created streams
- Instead of `_read` and `_write`, `Transform` streams require user implementation of the `_transform` method
- `_transform` combines both `_read` and `_write` APIs
  - the function signature is the same as `_write`
  - a `push` method exists on the instance (`this`), as with `_read`
  - the `done` callback can alternatively be used to push data. Pass data as the second param: `done(err, data)`

Creating Transform Streams

- `Transform` streams express an explicitly causal relationship between input and output
  - The `_transform` method enforces this relationship
  - However the relationship is loose, there doesn't have to be a one to one mapping between data, nor even between emitted chunks, neither regarding data size.

Transform Stream Walkthrough

```
$ node pipe1.js
```

- Transforms file contents to upper case
  - Loads the `fs` and `stream` modules
  - makes read stream from `pipe1.js` (`inStream`)
  - instantiates Transform stream as `upperStream`implements `_transform` method, which converts incoming `data` to upper case and pushes it out
  - `pipes` inStream through upperStream and to `process.stdout`

Streams Utility Belt

- Dominic Tarr (nearForms Head of Mad Science)
  - wrote the **event-stream** module
  - utility belt for streams:
    - **through**: easy transforms
    - **split**: split data into lines
    - **replace**: replace data in stream
    - **merge**: combine streams
    - **stringify**: convert data to strings
    - **parse**: convert JSON strings to objects
    - http://github.com/dominictarr/event-stream

nearForm

Through Stream Walkthrough

```
$ node pipe2.js
```

- Transforms file contents to upper case, using `event-stream`
  - Loads the `fs` and event-stream modules, assign event-stream to es
  - makes read stream from `pipe1.js` (`inStream`)
  - `pipes` inStream through the return value of `es.through` and to `process.stdout`
    - Callback passed to es.through simply takes a `data` param
    - `this.queue` is used in the same way as `push`

Stream Debugging

- Joyent's `vstream` module instruments streams with debug capabilities
  - Currently a young project, but as a development module it already offers some unique value
  - Analyse snapshots of stream buffers against `highWaterMarks`
  - Emit custom warnings
  - Trace where objects have come from in a pipeline of object mode streams
  - Automatic counters for object passing between streams and custom counters for tracking events
  - https://npmjs.org/vstream

Stream Debugging

```
var fs = require('fs');
var stream = require('stream');
var instream = fs.createReadStream('/usr/bin/more');
var passthru = new stream.PassThrough();
var outstream = fs.createWriteStream('/dev/null');

instream.pipe(passthru).pipe(outstream);
```

- Some stream based code prior to implementation
- Pipes a read stream from `/usr/bin/more` through a `PassThrough` stream and out into a write stream pointing a `/dev/null`

Stream Debugging

Instrumenting the streams

```
var vstream = require('vstream');
/*..other requires a stream instantiations..*/

vstream.wrapStream(instream, 'source')
vstream.wrapStream(passthru, 'passthru')
vstream.wrapStream(outstream, 'devnull')

instream.pipe(passthru).pipe(outstream);

instream.on('data', report);
outstream.on('finish', report);
function report() {
  //walk the pipeline
    instream.vsWalk(function (stream) {
      //output debug info on each stream
      stream.vsDumpDebug(process.stdout);
    });
    console.log('---')
}
```

nearForm

Stream Debugging Walkthrough

```
$ node debug-streams.js
```

- Requires the `vstream` module
- Wraps all three streams, assigning them logging names `'source'`, `'passthru'` and `'devnull'`.
- Registers report function with the `instream` `'data'` event and `outstream` `'finish'` event
- uses `vsWalk` method added by vstream to walk the pipeline
- calls `vsDumpDebug` on each stream in the pipeline

Circular Pipelines

```
duplex.pipe(duplex)

duplex.pipe(transform).pipe(duplex)
```

- Duplex streams can be piped back into themselves
- This enables two way communication
- It works because the readable and writable interfaces point to different targets
  - For instance, the readable interface points to an incoming network socket, the writeable points to an outgoing socket
  - Thus data doesn't infinitely cycle

Circular Pipelines Walkthrough

```
$ node echo0.js
```

- Creates a TCP server and echoes back any incoming data
- Loads the net module
- Instantiates a TCP server with net.createServer
- Calls listen on the server, setting port to 8124
- Pipes incoming connection sockets into themselves

```
$ telnet localhost 8124
```

Circular Transform Walkthrough

```
$ node echo1.js
```

- Creates a TCP server and echoes back uppercased input data
  - In addition to `net`, loads the `event-stream` module
  - Inserts a transform stream, created using `es.through` as it's being passed in to pipe.
  - The through stream does double duty
    - Uppercases input
    - Unless the input is `"quit"` in which case the `connection` is closed, allowing the session to be ended via telnet

```
$ telnet localhost 8124
```

Splitting Streams

```
readable.pipe(writable1)

readable.pipe(writable2)
```

- Attaching multiple pipes splits the pipeline
- If multiple pipes aren't attached at the same time (within the same event loop), the split will fail
  - Because data has continued to flow, the stream may have even ended
  - Splitting across event loops can be achieved by stream copying
    - Pipe to a `PassThrough` stream, use that pass through instance later on to pipe elsewhere (however this can have back-pressure implications)

Stream Splitting Walkthrough

```
$ node echo2.js
```

- Refactored through streams into `strStream`, `quitStream`, `upperStream` functions
  - Each returns a transform stream generated with `event-stream`
- Pipe connection through `strStream` to transform incoming binary data to strings. Assign the result of this to the stream variable

```
$ telnet localhost 8124
```

Stream Splitting Walkthrough

```
$ node echo2.js
```

- Split the stream by creating two pipelines
  - The first pipes to `process.stdout`
  - The second pipes to quitStream, upperStream and back into the connection

```
$ telnet localhost 8124
```

Real World Example

http://gosphero.com

- Drive a robot from the web browser
- Independently track and visualise the robots movement in the same browser window
- Code will use a fully stream-based design

nearForm

Browserify

```
$ sudo npm -g install browserify
```

- `browserify` packages code written in Node for use in browser
- It inlines dependencies - including core modules, like `stream`
  - Even seemingly back end specific modules like `fs` can be catered to with either `brfs` transform module or the `browserify-fs` replacement strategy module
- It supplies the require function, which pulls the inlined module into your modules context.
- More info at http://browserify.org

UI Streams

```
$ browserify public/js/front0.js >
  public/js/bundle0.js
```

- `front0.js` creates a logger stream for running in the browser
- Stream is tested with a `setInterval` writing to it every second
- browserify packages it into `bundle0.js`
  - inlines the stream module and its sub-dependencies alongside our `front0.js` code

Creating the Web Server

```
$ node web0.js
```

- `web0.js` uses express to create a web server
- Loads the express module and core http module
- Creates an express instance, assigned to `app`
- Uses the express.static middleware with `app.use`

Creating the Web Server

```
$ node web0.js
```

- Creates a server with `http.createServer` and passes `app` as the callback function, listens on port `3000`
  - Using `http.createServer` instead of calling `app.listen` frees us up to attach a web socket server later
- Open http://localhost:3000/index0.html

Websocket Streams

```
$ browserify public/js/front1.js > public/js/bundle1.js
$ node web1.js
```

- `front1.js` requires `websocket-stream` which wraps the native Websocket API in a stream
  - we set up a websocket stream pointing at our server (`web1.js`) and pipe its output into the logger stream
- `web1.js` uses the `ws` module to instantiate a web socket server (wss) that's attached to our webserver
  - When a web socket connection comes in (`ws`) we use `websocket-stream` to convert that web socket to a stream (wsStream)
  - Test our socket by calling write every second
- Open http://localhost:3000/index1.html

Streaming key input

```
$ node key0.js
```

- We'll need to read key input to drive the robot
  - Creates a `keypresser PassThrough` stream
  - Instead of piping listen for data events and write them to keypresser
    - Just as proof of concept, in the next step we'll be listening to a non-stream `EventEmitter`

```
$ pkill node
```

Streaming key input

```
$ node key0.js
```

- Pipe the keypresser to `process.stdout`
- `process.stdin` is paused by default, we have to call `resume` on it - this prevents the process from exiting, `setRawMode` allows input like arrow keys.

```
$ pkill node
```

nearForm

Object Mode Streams

```
$ node key1.js
```

- Requires the `event-stream` and `keypress` modules
- Make the `keypresser` an object stream
- Pass `process.stdin` to third party keypress module
  - Causes process.stdin to emit custom `'keypress'` events

Object Mode Streams

```
$ node key1.js
```

- Listen to 'keypress' events, write objects containing `ch` and key params to the keypresser stream
- pipe keypresser through `es.stringify` (essentially `JSON.stringify` as a stream) and into `process.stdout`

Transforming Keys into Commands

```
$ node key2.js
```

- Adds a `Transform` stream, that converts key objects into command objects, reinstates `Ctrl+C`
  - Creates the `keyparser` stream in object mode and implements the `_transform` method, which builds a `cmd` object based on the incoming `keypress` object then passes that command to `done`
  - A `killer` stream is also created with `es.through`, this will exit the process when it sees a `'die'` command.
  - We pipe the `keypresser` into keyparser through the killer, through `es.stringify` and out to `process.stdout`

Making a Controller Stream

```
$ node control0.js
```

- Evolution of `key2.js` - Adds a `Controller Transform` stream calculates and manages state, attaching state to the `cmd` and holding it internally
- Rather than creating a Transform stream directly, the Controller inherits from Transform
- Controller is instantiated at the top along with the other streams, assigned to `controller`
- Pipe from `keypresser` through `keyparser` through `killer` through controller through `es.stringify` and out to `process.stdout`

Convenience Testing Stream

```
$ node control1.js
```

- Adds a `RandomLocator Duplex` stream which generate random movement events
  - Calls `setInterval` upon instantiation, to `push` a random location command every second
- We'll use bluetooth to talk to the real robot
  - But that's slow for development work

Connecting it all together

```
$ browserify public/js/front2.js > public/js/bundle2.js
$ node web2.js
$ node control2.js
```

- `front2.js` is the same as `front1.js`
- `web2.js` adds a TCP server, pipes commands coming over TCP into WebSocket, logging them out to `process.stdout`
- Creates a TCP client, connects it to `web2.js` TCP server, splits the pipeline into TCP client and `process.stdout`
- Open http://localhost:3000/index2.html

Streaming to Canvas

```
$ browserify public/js/front3.js > public/js/bundle3.js
$ node web3.js
$ node control3.js
```

- `web3.js === web2.js` and `control3.js === control2.js`
- `front3.js` shows movement graphically, streaming input to the UI
  - Uses FabricJS, HTML canvas library (http://fabricjs.com)
  - Modify `logger._write` method to draw incoming location data onto the fabric.Canvas
  - The location data comes in over the `websocket-stream` and is piped into logger
  - `logger` draws the input

Stream Browser Keys to Server

```
$ browserify public/js/front4.js > public/js/bundle4.js
```

- Send key commands from the browser to the server, to allow us to drive Sphero from either the terminal or the ui
- Create `keystream`, similar to the `keypresser` stream on the server, but not in object mode (`websocket-stream` doesn't support object mode)
- sends key presses over the websocket stream using `keymaster` library (http://github.com/madrobby/keymaster)

Stream Browser Keys to Server

```
$ browserify public/js/front4.js > public/js/bundle4.js
```

- creates a global key variable, which is used to listen to a particular key presses
- we set up listeners for keys we're interested in, each listener writes a command based on the key into the keystream

Handle Incoming UI Key Stream

```
$ node web4.js
$ node control4.js
```

- `web4.js` pipes from cmdStream to wsStream through a logging stream and then back into cmdStream
- `control4.js` adds a pipe from client through `es.parse` (covering any strings coming from `websocket-stream`) to the `keypresser`
- Commands generated from key presses `control4.js` terminal get sent over web socket to browser
- Commands generated key presses in the ui get piped into the `cmdStream`, and thus through the pipeline in `control4.js` then back over across to websocket stream to the browser
- Open http://localhost:3000/index4.html

sphero.js

```
$ node web4.js
$ node sphero.js
```

- We have to make sure that Sphero is on and bluetooth paired first
- `sphero.js` is an evolution of `control4.js`
- removes the `RandomLocator` stream
- loads `node-sphero` module, assigns it to `roundRobot`, creates an instance of `roundRobot.Sphero`, assigns it to `sphero` variable.

sphero.js

```
$ node web4.js
$ node sphero.js
```

- Adds a `spheroEvents PassThrough` stream
- Adds a `locprinter` stream with `es.through`, this presents location information on the terminal when included in the pipeline
- Adds a `SpheroDrive` stream that inherits from `Duplex`
  - Converts command objects into relevant invocations on the `sphero` object

sphero.js

```
$ node web4.js
$ node sphero.js
```

- When node process starts `sphero.connect` is called
- We listen for a `'connect'` event, which is passed a `ball` argument
- Perform some initial setup on the ball
- Register a `'notification'` listener on the ball, write to `spheroEvents` based location updates

sphero.js

```
$ node web4.js
$ node sphero.js
```

- this means that if we physically control the ball (e.g. kick it), the canvas will update position
- Creates `spheroDrive`, and instance of our `SpheroDrive Duplex` stream
- Sets up the pipeline:
  `keypresser | keyparser | spheroEvents | killer | controller | spheroDrive | locprinter | es.stringify | client`

Where Next?

- Read the core documentation:
  - http://nodejs.org/api/stream.html
  - http://nodejs.org/api/buffer.html
- Do the stream adventure:
  - http://nodeschool.io/#stream-adventure
- Node Cookbook, Chapter 5: Employing Streams
  - http://amzn.com/1783280433