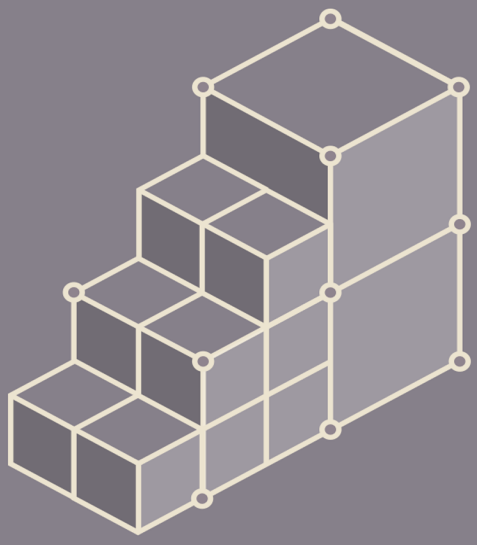
YLD publishing presents



FLOW CONTROL

MASTERING ASYNCHRONOUS
FLOW CONTROL IN NODE

Pedro Teixeira

Flow Control Patterns

Mastering asynchronous flow control in Node

Pedro Teixeira

This book is for sale at http://leanpub.com/flowcontrolpatterns

This version was published on 2016-02-10

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. The source code

You can find the source code for the entire series of books in the following URL:

https://github.com/pgte/node-patterns-code[1]

You can browse the code for this particular book here:

https://github.com/pgte/node-patterns-code/tree/master/02-flow-control-patterns[2]

You are free to download it, try it and tinker with it. If you have any issue with the code, please submit an issue on Github. Also, pull requests with fixes or recommendations are welcome!

[1] https://github.com/pgte/node-patterns-code
[2] https://github.com/pgte/node-patterns-code/tree/master/02-flow-control-patterns

2. Introduction

Given its async nature, JavaScript code usually relies on functions for flow control when doing I/O. Typically, if you have a function that has to wait on a result – say, a response from a server in the network – you must provide a function to be called back. For instance, if you're doing some browser jQuery code to perform an HTTP GET request, you could do:

```
$.get('http://some.server.com/some/url', function success(data) {
  // do something
});
```

But what happens when the request, for some reason, is not able to succeed? If you wanted to get notified of the errors you would have to fall back on the more powerful `$.ajax` method:

```
var settings = {
  success: function success(data) {
    // do something
  },
  error: function error(req, status, error) {
    // handle error
  }
};

$.ajax('http://some.server.com/some/url', settings);
```

In the error-handling function, in order to understand more details about the origin of the error, you would have to parse the `status` argument of the error callback, which can have the values "timeout", "error", "abort", "parseerror" or even `null`.

One of the problems with this interface is that it's hard to compose. If you wanted to perform another HTTP request right after this request succeeded, your code would look indented and hard to read:

```
var settings = {
  success: function success(data) {
    var settings2 = {
      success: function success2(data2) {
        // success, handle the results data and data2
      },
      error: function error2(req, status, error) {
        // handle error 2
      }
    };

    $.ajax('http://some.server.com/some/other/url', settings2);
  },
  error: function error(req, status, error) {
    // handle error
  }
};

$.ajax('http://some.server.com/some/url', settings);
```

To make this more manageable we could decrease two levels of indentation by not inlining the callback functions:

```
function handleSuccess() {

  function handleSuccess2(data2) {
    // success, handle the results data and data2
  }

  function handlError2(req, status, error) {
    // handle error 2
  }

  var settings2 = {
    success: handleSuccess2,
    error: handlError2
  };

  $.ajax('http://some.server.com/some/other/url', settings2);
}

function handleError(req, status, error) {
```

```
  // handle error
}

var settings = {
  success: handleSuccess,
  error: handleError
};

$.ajax('http://some.server.com/some/url', settings);
```

But still, simple I/O is obviously hard to compose, and we have yet to handle an occurring error.

What if the second call was not dependent on the result of the first one? You could do them in parallel, and the code to handle them would look something like this:

```
function handleSuccess(responses) {
  // handle responses per URL here
}

function handleError() {
  // handle error
}

var urls = [
  'http://some.server.com/some/url',
  'http://some.server.com/some/other/url'
];

var responses = {};

checkFinished() {
  if (Object.keys(responses) == urls.length) {
    handleSuccess(responses);
  }
}


urls.forEach(function(url) {
  function handleSuccess(data) {
    responses[url] = data;
    checkFinished();
  }
```

```
  var settings = {
    success: handleSuccess,
    error: handleError
  };

  $.ajax(url, settings);
});
```

That's a lot of machinery needed just to make a couple of parallel requests. What patterns could make handling I/O easier?

3. The Callback patterm

Unlike the previous example that used jQuery where there may be several callback functions involved in one call — one for error and one for success — there's a pattern that Node.js and a lot of third-party libraries use to make it easy to compose different types of I/O calls.

This pattern consists of these principles:

- One callback to rule them all: one sole function instead of a success and an error one;
- The callback is called once and exactly once. It's called when an error occurs, or a result is available;
- Function-last: the function is the last argument of the call that initiates I/O;
- Error-first: the callback has the following function signature:

```
function (err, result) {
  //...
}
```

The values passed into the callback have different values depending on whether an error occurred or not:

If no error occurred:

- The first callback argument is `null` or `undefined`;
- The second argument contains the callback results.

On the other hand, if an error occurred:

- the first argument contains an error object, an instance of the JavaScript `Error` class (or sub-class) that describes the error;
- the second argument will be undefined.

This patterns forces the programmer to not ignore errors (by not defining an error callback) and, more importantly, defines a common pattern on top of which you can build abstractions and utilities.

Here is an example of the callback pattern in action in Node.js when reading the contents of a file:

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

fs.readFile('/some/path/to/file.txt', function(err, fileContent) {
  if (err) {
    handleError(err);
  } else {
    // use fileContent
  }
});

function handleError(err) {
  console.error('error caught while reading file:', err);
}
```

Here we're handling occurring errors simply by logging into the console, but depending on your application you may want to handle it more appropriately: log it into a central place, notify someone, terminate a request, etc.

The important part of the callback function is that we're breaking the flow of the application. Instead of using an `if/else` branch for that, you may also use a return, which would go like this:

```
fs.readFile('/some/path/to/file.txt', function(err, fileContent) {
  if (err) {
    return handleError(err);
  }
  // use fileContent
});
```

4. Orchestrating callbacks

It's not only Node core API that uses the callback pattern: a multitude of third-party libraries that live on NPM also use this pattern. This allows you to use a flow-control library like `async` to compose them in almost any way you would want.

To use async you must install it in the root directory of your code using:

```
$ npm install async
```

4.1 Series

The simplest case of I/O orchestration is the case where you want to chain a series of I/O calls together, one after another, and interrupt them if an error occurs.

This example runs two operations, each of them removing one file:

async_series_1.js:

```
const async = require('async');
const fs = require('fs');

var work = [removeFile1, removeFile2];

function removeFile1(cb) {
  fs.unlink('./file1.txt', cb);
}

function removeFile2(cb) {
  fs.unlink('./file2.txt', cb);
}

async.series(work, done);

function done(err) {
  if (err) throw err;
  console.log('done');
}
```

The `async.series` function gets two arguments: a first argument with an array of functions that are to be invoked in sequence. These functions (our `removeFile1` and `removeFile2`) have to have the same signature: a callback function as the first and sole argument. Then the second argument of `async.series` is a final callback that gets called when all of the functions terminate, or when one function gets an error.

If you place two files named `file1.txt` and `file2.txt` in the current directory, you will be able to run this without an error:

```
$ node async_series_1.js
```

If you run this again you should get an error on the `done` callback because the `file1.txt` was removed:

```
  if (err) throw err;
                ^
Error: ENOENT, unlink './file1.txt'
```

Here you can see that the series flow breaks the first time an error occurs, and in this case the final callback function (`done`) gets called with an error in the first arguments.

4.2 Parallel

If the I/O you're doing is independent, you can reduce the total time a set of operations take by doing them in parallel.

```
const async = require('async');
const fs = require('fs');

var work = [removeFile1, removeFile2];

function removeFile1(cb) {
  fs.unlink('./file1.txt', cb);
}

function removeFile2(cb) {
  fs.unlink('./file2.txt', cb);
}


async.parallel(work, done);
```

```
function done(err) {
  if (err) throw err;
  console.log('done');
}
```

The only thing we changed here was to replace `async.series` with `async.parallel` to cause these two I/O operations to be performed in parallel.

If one of these operations fails, the first one to fail triggers the `done` callback with an error. If any of the pending calls finish, the `done` callback doesn't get called. (Remember: a callback function is never called twice; this is part of the contract.)

This is one of the drawbacks of using I/O in parallel instead of in series: you're only able to handle the first error that happens. If you need to handle each one of them, either avoid parallel calls using `async.series` or have one custom callback to handle the error on each individual operation.

4.3 Waterfall

The reason for using `async.series` would be that the I/O calls were dependent. Imagine that first you need to fetch a value from a remote system like a database server in order to be able to create the next request. Here is such an example:

```
function befriend(user1, user2, cb) {

  var alreadyFrends = false;

  async.series([findIfAlreadyFriended, friend], cb);

  function findIfAlreadyFriended(cb) {
    Friends.exists({from: user1, to: user2}, function(err, areFriends) {
      if (err) {
        cb(err);
      }
      else {
        alreadyFriends = areFriends;
      }
    });
  }
```

```
  function friend(cb) {
    if (alreadyExists) {
      return cb();
    }

    Friends.insert({from: user1, to: user2}, cb);
  }
}
```

Notice that we're using the `befriend` third argument, a callback function, as a direct last argument of the `async.series` call. Here you can already see a big advantage of the callback pattern: you don't have to adapt between different function signature patterns – you can pass the callback function directly as an argument.

This `befriend` function creates a "friend" link in a remote database. It's using `async.series` to compose these two functions: the first finds out whether the link already exists, and the second creates the friendship link, but only if that link doesn't already exist.

The `findIfAlreadyFriended` function then has to check the remote system to find out if such a link exists, and update the global `alreadyFriends` Boolean variable. This Boolean variable is then used in the next function in list, the `friend` function, to decide whether to do the link insertion or to callback immediately.

We designed this `befriend` function to not error out if such a friend link already exists. You could also design it to callback with an error, which would break the `async.series` flow as we wanted.

You can entirely avoid keeping a shared variable between functions by using the `async.waterfall` method:

```
function befriend(user1, user2, cb) {

  async.waterfall([findIfAlreadyFriended, friend], cb);

  function findIfAlreadyFriended(cb) {
    Friends.exists({from: user1, to: user2}, cb);
  }

  function friend(alreadyExists, cb) {
    if (alreadyExists) return cb();
```

```
    Friends.insert({from: user1, to: user2}, cb);
  }
}
```

Knowing that the `Friends.exists` function already calls back with a signature of `(err, exists)`, we use that to our advantage: we let `async.waterfall` pass whatever the results of the callback are into the next function, keeping the state you need in the call stack instead of the outer scope.

As in all the async flow control directives, `async.waterfall` also breaks on the first error, handing it off to the final callback.

Watch here that the `friend` function always has to invoke the `cb` callback no matter what. If we don't, the `befriend` callback will not get invoked in some cases. This is one of the hardest parts of using callbacks: making sure they always get invoked exactly once, no matter what.

4.4 Collections

If, instead of having a set of different functions, you want to apply the same function to a set of objects, you can use async's collection iteration function utilities.

4.4.1 Parallel iteration

As an example let's say that, given a list of documents, you want to insert all of the documents into this remote database:

```
var docs = // an array of documents

async.each(docs, insertDocument, done);

function insertDocument(doc, cb) {
  db.insert(doc, cb);
}

function done(err) {
  if (err) {
    handleError(err);
  }
  else {
    console.log('inserted all documents');
  }
}
```

Here we see that `async.each` accepts three arguments:

1. The collection (a JavaScript array)
2. The iterator function
3. A final callback function, called when all iterations are complete, or on the first occasion that an error occurs.

Our iterator function `insertDocument` will then be called by async for each document, in parallel. This function is responsible for accepting one object as the first argument, and a callback function as the last argument; this last to be called when there is an error, or this particular operation terminates.

Given that our `insertDocument` is only calling one function, we could use JavaScript `Object#bind` and reduce our example to:

```
var docs = // an array of documents

async.each(docs, db.insert.bind(db), done);

function done(err) {
  if (err) {
    handleError(err);
  }
  else {
    console.log('inserted all documents');
  }
}
```

4.4.2 Parallel, but with limit

The `async.each` function is quite handy, but is only useful if we know that the maximum length of the collection is relatively small. If the collection is too large, we risk:

- **overloading the receiving system**. For instance if, as in this example, we're invoking this procedure on a remote system, too many of those in parallel will risk overburdening that system. Worse still, if many of these operations are done, each one when answering a client request, we multiply this effect by the number of parallel client requests.
- **overloading Node.js memory**. Each of these calls inevitably allocates some resources on our Node.js process: some structures, closures and file descriptors. These resources will eventually be closed once the operation finishes, or will be garbage-collected in the near future – but too many of these in parallel and we risk putting too memory much pressure, eventually leading to resource exhaustion.

- **blocking the event loop**. Each one of the calls to start one of these operations blocks the event for a short period of time. Multiply that period of time by the number of objects you're iterating on, and you may be blocking the event loop for a long time, possibly degrading the experience for other users currently being served by your Node.js process.

To prevent this you can either cap the collection, thus losing documents, or you can simply use `async.eachLimit`, which allows you to impose a limit on the number of outstanding operations:

```
var docs = // an array of documents

var parallelLimit = 5;

async.eachLimit(docs, parallelLimit, db.insert.bind(db), done);

function done(err) {
  if (err) {
    handleError;
  }
  else {
    console.log('inserted all documents');
  }
}
```

By using `async.eachLimit` here, we're defining that we're allowing a maximum of five outstanding operations: that is, at any time, there is a maximum of five ongoing document insert calls. In this way you can help reduce the pressure on an external service, as well helping to reduce the pressure on memory and other resources.

Bear in mind, though, that this technique doesn't reduce the **overall** pressure that your Node.js process puts on an external service. If, for instance, you're going to perform this operation for every user request, even a local parallel limit of five like this one may not be enough, since the effect must be calculated by multiplying the parallel limit by the number of requests being served at any given time. (To reduce that, you can use other patterns such as a global queue or a connection pool).

Note that the time that it takes for a client request to complete may increase as you increasely limit the maximum outstanding parallel operations, which in turn may increase the number of pending parallel requests. This number can be fine-tuned according to your application load patterns.

5. Using a work queue

If an external service requires a global maximum of parallel requests, you can concentrate all the work behind a work queue using `async`. You may want to do this for several reasons:

- To reduce the pressure on a given external system
- To do perform asynchronous work

Async lets you define a queue, given a work function and a maximum number of outstanding requests. For instance, you can define a Singleton logger module like this:

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

var parallelLimit = 5;

var q = async.queue(sendLog, parallelLimit);

function sendLog(entry, cb) {
  sendLogEntryToExternalSystem(entry, cb);
}

module.exports = push;

function push(entry, cb) {
  q.push(entry, cb);
};
```

Here, on line 5 we're creating a queue, passing in a worker function (`sendLog`) and the maximum number of ongoing operations (five in this case). This singleton module then exports a function (`push`) that pushes the log into the queue using the `queue.push` method. This method accepts the payload as the first argument, and an optional callback as the second.

As the callback is optional, it allows the clients of this module not to care about the outcome of this operation:

```
var log = require('./logger');

log('payload 1');
log('payload 2');
```

If interested, as an alternative, the clients may pass in a callback function to get notified of the completion of each operation:

```
var log = require('./logger');

log('payload 1', handleLogDone);
log('payload 2', handleLogDone);

function handleLogDone(err) {
  if (err) {
    console.error('error logging: ' + err.stack);
  }
  else {
    console.log('logging finished well');
  }
}
```

This last feature can be used to provide a transparent queue in front of a service. Let's say, for instance, that you have a legacy database that is not very powerful and not particularly scalable, and that you must use it to insert documents. You can absorb peak traffic by putting this fragile service behind an async queue:

document_inserter.js:

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

var q = async.queue(insertDocument, 1);

function insertDocument(doc, cb) {
  fragileRemoteSystem.insertDocument(doc, cb);
}

module.exports = function push(doc, cb) {
  q.push(doc, cb);
};
```

Bear in mind that pushing an object into the queue occupies memory. A in-memory queue like this one is great to absorb peaks, but if the consuming rate is slower than the producing rate for a long time, your Node.js process will collapse due to memory exhaustion. To solve this you can alternatively a) increase the maximum allowed pending operations or b) use a persistent queue.

6. Using an event emitter

The callback pattern works well for simple operations that have a start and an end state; but if you're interested in state changes, a callback is not enough. Fortunately Node.js ships with an event emitter implementation. The event emitter pattern allows you to decouple the producers and consumers of events using a standard interface.

6.1 Consuming events

Let's say that your Node.js process is somehow able to connect to a home automation system that can notify you of several events happening in your home. If the system can notify you of the main doorbell ringing, a door object could emit an event named "bell". Any module could subscribe to doorbell rings by listening to that event on a given object:

```
var home = require('./home');

home.door.on('bell', function() {
  /// do something
});
```

In this case the `home.door` object is an event emitter. As a client of this object, you start listening to "bell" events by using the `.on` method and passing in a function that gets called whenever an event with that specific name happens.

Events can also bear some data. For instance, let's say that you want to get notified when a light bulb is turned on or off:

```
home.lights.on('on', function(number, room) {
  console.log('light number %d on room %s was turned on', number, room);
});

home.lights.on('off', function(number, room) {
  console.log('light number %d on room %s was turned off', number, room);
});
```

Here you can see that the "on" and the "off" events bear the room name. This is arbitrary data that is emitted by the emitter.

Besides subscribing to events, you can also unsubscribe by using `listener.removeListener`, specifying the event type and the event listener function. To use it you will have to name the listener function like this:

```
function onTurnLightOn(number, room) {
  console.log('light number %d on room %s were turned on', number, room);
}

home.lights.on('turnon', onTurnLightOn);

/// later...

home.lights.removeListener('turnon', onTurnLightOn);

// `onTurnLightOn` doesn't get called after this
```

6.2 Producing events

You can use the event emitter as a producer by constructing a JavaScript pseudo-class and use Node's built-in `util.inherits` to setup the proper inheritance attributes. Here is an example of a `clock` event emitter that emits `tic` and `toc` events:

clock_emitter_class.js:

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

module.exports = Clock;

function Clock() {
  if (! (this instanceof Clock)) return new Clock();

  this._started = false;

  EventEmitter.call(this);
}

inherits(Clock, EventEmitter);

Clock.prototype.start = function start() {
  var self = this;

  if (self._started) return;

  var tic = true;
```

```
  this._started = Date.now();

  self._interval = setInterval(function() {
    var event = tic ? 'tic' : 'toc';
    self.emit(event, self.time());
    tic = ! tic;
  }, 1000);
};

Clock.prototype.stop = function stop() {
  clearInterval(this._interval);
  this._started = false;
};

Clock.prototype.time = function() {
  return this._started && Date.now() - this._started;
};
```

Here you can see that the clock itself is an event emitter, and it's internally calling `self.emit` to emit arbitrary events (`tic` and `toc` in this case). Alongside each event, we're passing the current clock time (in milliseconds since the time it was started) by using the arguments in `self.emit` following the event name.

Clients of this Clock class can then do:

```
var Clock = require('./clock');
var clock = Clock();

clock.on('tic', function(t) {
  console.log('tic:', t);
});

clock.on('toc', function(t) {
  console.log('toc:', t);
});

clock.start();

// stop the clock 10 seconds after
setTimeout(function() {
  clock.stop();
}, 10e3)
```

6.2.1 How event emitter handles errors

As we've seen, an event type is defined by an arbitrary string. In that aspect, all events are treated equally, with one exception: errors.

If an event emitter emits one event for which it has no listeners, that event gets ignored. But, in the special case of the event name being "error", the error is instead thrown into the event loop, generating an uncaught exception.

If that happens, you can catch an uncaught exception by listening to the `uncaughtException` that the global `process` object emits:

```
process.on('uncaughtException', function(err) {
  console.error('uncaught exception:', err.stack || err);
  // orderly close server, resources, etc.
  closeEverything(function(err) {
    if (err)
      console.error('Error while closing everything:', err.stack || err);

    // exit anyway
    process.exit(1);
  });
});
```

When catching an uncaught exception, you should treat it as a permanent failure in your Node process: consider taking this opportunity to close the services in an orderly fashion, and eventually terminate the process.

Yes, shutdown your process when an uncaught exception occurs. Otherwise you will most probably end up with a resource leak due to the unknown state some Node objects will be in.

6.3 Listening to events once

What happens when, in the above example, more than one uncaught exception gets caught? Both will trigger the `closeEverything` function, which may bring problems on your shutdown procedure. In this case you should probably only consider the first instance of an uncaught exception happening. You can do that by using `emitter.once`:

```
process.once('uncaughtException', function(err) {
  // orderly close server, resources, etc.
  closeEverything(function(err) {
    if (err)
      console.error('Error while closing everything:', err.stack || err);

    // exit anyway
    process.exit(1);
  });
});
```

But now what happens when two `uncaughtExceptions` occur? The second one will find no event listener, and will trigger an immediate process shutdown, interrupting our beloved `closeEverything` that was taking care of an orderly shutdown.

To avoid that we can still log every uncaught exception we get meantime by adding a second listener:

```
process.on('uncaughtException', function(err) {
  console.error('uncaught exception:', err.stack || err);
});
```

6.4 Synchronous event delivery

Node.js is asynchronous, but since no I/O is envolved in emitting an event, event delivery is treated synchronously.

emit_asynchronous.js:

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

var emitter = new EventEmitter();

emitter.on('beep', function() {
  console.log('beep');
});

emitter.on('beep', function() {
  console.log('beep again');
});

console.log('before emit');
```

```
emitter.emit('beep');

console.log('after emit');
```

If run this file you will get the following output:

```
before emit
beep
beep again
after emit
```

So, when emitting events, bear in mind that the listeners will be called before `emitter.emit` returns.

Remembering this last bit can save you a lot of debugging time when using more than one listener for the same event.

6.5 Event delivery exceptions

Since the event delivery is synchronous to the emission, what happens when one listener accidently throws an exception? Let's see:

emit_throws.js:

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

var emitter = new EventEmitter();

emitter.on('beep', function() {
  console.log('beep');
});

emitter.on('beep', function() {
  throw Error('oops!');
});

emitter.on('beep', function() {
  console.log('beep again');
});

console.log('before emit');
```

```
try {
  emitter.emit('beep');
} catch(err) {
  console.error('caught while emitting:', err.message);
}

console.log('after emit');
```

Running the code above will produce the following output:

```
before emit
beep
caught while emitting: oops!
after emit
```

This is bad news. It means that, when emitting events, if one of the listeners throws, depending on the order of the listener registration, the listener may not be notified. This means that the event emitter is a good pattern for *logically* decoupling producers and consumers at the API level, but in practice the event producers and consumers are somewhat coupled.

For a more decoupled approach you can use an in-memory queue, or even a persistent queue (discussed in another book of this series).

7. Streams

The event emitter is a convenient pattern for getting notifications of complex state changes in a decoupled manner, but some objects are simpler than that: they're a source of data or the target of data.

7.1 Readable stream

In this article we're going to continue building on our home automation system. Our system has a thermometer that frequently emits temperature readings. This thermometer acts like a data tap: what Node.js normally calls a Readable Stream.

Here is how we could listen to the temperature readings from that thermometer:

```
var thermometer = home.rooms.living_room.thermometer;

thermometer.on('data', function(temperature) {
  console.log('Living room temperature is: %d C', temperature);
});
```

Here you can see that a readable stream is a special case of an event emitter, emitting `data` events every time some new data is available.

A readable stream can emit any type of data: binary data in the form of buffers or strings, or even more complex data in the form of any JavaScript object.

In addition to emitting data, a readable stream can be paused and resumed:

```
thermometer.pause();

// in 5 seconds, resume monitoring temperature:
setTimeout(function() {
  thermometer.resume();
}, 5000);
```

When paused, a readable stream will emit no more data until resumed.

There are several examples of readable stream implementations in the Node.js and outside of it. Here are some of them:

- The contents of a file
- An server HTTP request body
- A server TCP connection
- A client TCP connection
- An client HTTP response body
- The changes on a database
- A video stream
- An audio stream
- The results of a database query
- and many more...

7.1.1 Creating a readable stream

There are several ways of creating a readable stream. One way is to inherit from Node.js `stream.Readable`, implementing the `_read` method.

thermometer.js:

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

module.exports = Thermometer;

function Thermometer(options) {
  if (! (this instanceof Thermometer)) return new Thermometer(options);
  if (! options) options = {};
  options.objectMode = true;
  Readable.call(this, options);
}

util.inherits(Thermometer, Readable);

Thermometer.prototype._read = function read() {
  var self = this;

  getTemperatureReadingFromThermometer(function(err, temperature) {
    if (err) self.emit('error', err);
    else self.push(temperature);
  });
};
```

Our thermometer constructor initialises itself by calling the super-class constructor on itself, assuring that `options.objectMode` is true. This enables the stream to handle data other than strings or buffers, as is our case.

Besides inheriting from `stream.Readable`, our readable stream has to implement the `_read` method, which is called when the stream is ready to pull in some data. (More about this data pulling later.) This method then fetches the needed data (in our case, from the hardware temperature sensor) and, once available, gets pushed into the stream by using `stream.push(data)`.

7.1.2 Pull stream vs. push stream

There are two main types of read stream: one that you must pull data from, and one that pushes data onto you. Am illustration of a push stream is a water tap: once you open it, it keeps gushing water. An illustration of a pull stream can be a drinking straw: it pulls the water only when the straw user sucks on it.

> Another, and perhaps a more apt, real-world analogy of a pull stream is a bucket brigade. If there is nobody there to take the bucket, the bucket holder is just left standing there waiting.

Node.js core streams have these two modes. If you simply listen for the `data` event, the push mode is activated and data flows as fast as the underlying resources can push them:

thermometer_client_push.js:

```
var Thermometer = require('./thermometer');

var thermomether = Thermometer();

thermomether.on('data', function(temp) {
  console.log('temp:', temp);
});
```

If, instead, you read from the stream, you're using the default pull mode and reading at your own rate, as in this example:

```
var Thermometer = require('./thermometer');

var thermometer = Thermometer({highWaterMark: 1});

setInterval(function() {
  var temp = thermometer.read();
  console.log('temp:', temp);
}, 1000);
```

This client uses the `stream.read()` method to get the latest reading from the stream. Notice that we initialise the stream with a `highWaterMark` option value of 1. When the stream is in *object mode*, this setting defines the maximum number of objects that our stream buffers. As we want to minimise the possibility of getting stale temperature data, we only keep a maximum of one reading buffered.

> When we introduce `stream.pipe` for hooking streams together, you will see where these values, and the way data flows, come into play.

7.2 Writable streams

There's another type of stream that, instead of emitting data, can have data sent to it: the writable stream. Some examples of writable streams in Node are:

- a writable file in append mode
- a TCP connection
- the process standard output stream
- a server HTTP response body
- a database bucket (or table)
- an HTML parser
- a remote logger
- ... and many more

To write into a writable stream you simply call `stream.write(o)`, passing in the data you're writing. You can also pass in a function that gets called once that payload gets flushed: `stream.write(payload, callback)`.

To implement a writable stream you can inherit from Node.js `stream.Writable` and implement the protected `stream._write` method.

As an example, let's say that you want to write every single thermometer reading into this database server, but the database layer module you have to use does not provide a streaming API. Let's then implement one:

db_write_stream.js:

```
var Writable = require('stream').Writable;
var util = require('util');

module.exports = DatabaseWriteStream;

function DatabaseWriteStream(options) {
  if (! (this instanceof DatabaseWriteStream))
    return new DatabaseWriteStream(options);
  if (! options) options = {};
  options.objectMode = true;
  Writable.call(this, options);
}

util.inherits(DatabaseWriteStream, Writable);

DatabaseWriteStream.prototype._write = function write(doc, encoding, callback) {
  insertIntoDatabase(JSON.stringify(doc), callback);
};
```

The implementation of this write stream is really simple: besides the inheritance ceremony and enabling object mode, the only relevant thing here is to define the `_write` function that performs the actual write into the underlying resource (our document database). The `_write` function must accept three arguments:

- `chunk`: the piece of data you're writing. In our case, it's the document we're about to insert into the database.
- `encoding`: if the data is a string (not our case), what is the encoding?
- `callback`: a callback function to be called once the write finishes or errors.

We can now use this stream to write thermometer data into:

```
var DbWriteStream = require('./db_write_stream');
var db = DbWriteStream();

var Thermometer = require('./thermometer');

var thermomether = Thermometer();

thermomether.on('data', function(temp) {
  db.write({when: Date.now(), temperature: temp});
});
```

OK, this is nice – but why did we bother creating a writable stream on top of our database? Why not just stick with the original database module? Because streams are composable by using `stream.pipe()`.

7.3 Connecting streams using pipe

Instead of manually connecting the two streams, we can connect a readable stream into a writable stream using `stream.pipe` like this:

```
var DbWriteStream = require('./db_write_stream');
var db = DbWriteStream();

var Thermometer = require('./thermometer');
var thermomether = Thermometer();

thermomether.pipe(db);
```

This keeps the data flowing between our thermometer and our database. If, later in time, we want to stop the flow, we can disconnect both streams using unpipe:

```
// Unpipe after 10 seconds

setTimeout(function() {
  thermometer.unpipe(db);
}, 10e3);
```

The advantage of `pipe` is not only to avoid us from writing code to transport the stream chunks from one stream into another: it also gives us flow control for free.

7.4 Flow control in streams

When you hook a readable stream and a writable stream together with `readable.pipe(writable)`, the flow of data is adjusted to the consumer speed. Under the hood, pipe uses the writable `.write()` method, and uses the method return value to decide whether or not to pause the source stream like this: if the `writable.write` returns `true`, it means that the written data was flushed into the underlying resource — our database in this specific case. If the call returns `false`, it means that the writable stream is buffering the write, which means that the source stream needs to be paused. Once the writable stream is drained, it appropriately emits the `drain` event, signalling the pipe to resume the source stream.

Also, and as we've touched on briefly, you can control the maximum amount of buffering that a readable stream does by defining the `options.highWaterMark` value. If the stream is a binary one (also sometimes called a "raw" stream), this value represents the maximum amount of bytes that the stream is willing to buffer. If the stream is in object mode, it represents the maximum number of objects that it's willing to buffer.

As an implementor of any type stream, you don't have to worry about the implementation of this. When implementing a writable stream, you just have to callback when the given chunk or object is sent. When using a readable stream, the stream client can define the `options.highWaterMark` value to control the maximum amount of buffering. All the rest is handled by the Node.js streams' implementation.

7.5 Transform streams

We've looked at readable and writable streams, but there is a third type of stream that combines these two: it's called a *Through* or *Transform* stream.

Streams are not necessarily meant to be used in pairs: one readable stream piped into a writable stream. They can be composed into this pipeline where the data flows from a readable stream into one or more *transform streams* and ends up in a writable stream.

In our database writable stream example, we accept JavaScript objects and transform them into a JSON string before inserting them in the database. Instead we can create a generic transform stream that does that:

json_encode_stream.js:

```
var Transform = require('stream').Transform;
var inherits = require('util').inherits;

module.exports = JSONEncode;

function JSONEncode(options) {
  if ( ! (this instanceof JSONEncode))
    return new JSONEncode(options);

  if (! options) options = {};
  options.objectMode = true;
  Transform.call(this, options);
}

inherits(JSONEncode, Transform);

JSONEncode.prototype._transform = function _transform(obj, encoding, callback) {
```

```
  try {
    obj = JSON.stringify(obj);
  } catch(err) {
    return callback(err);
  }

  this.push(obj);
  callback();
};
```

To implement a transform stream we inherit from Node's `stream.Transform` pseudo-class and implement the protected `_transform` method, which accepts the raw JavaScript object and pushes in a JSON-encoded string representation of it.

Also, you may have noticed that, when we introduced pipes, we gave up on pushing documents with timestamps into the database; instead we pushed raw temperature readings. We can correct this now by introducing another transform stream that accepts a temperature reading and spits out a time-stamped document:

to_timestamped_document_stream.js:

```
var Transform = require('stream').Transform;
var inherits = require('util').inherits;

module.exports = ToTimestampedDocTransform;

function ToTimestampedDocTransform(options) {
  if ( ! (this instanceof JSONTransform))
    return new JSONTransform(options);

  if (! options) options = {};
  options.objectMode = true;
  Transform.call(this, options);
}

inherits(ToTimestampedDocTransform, Transform);

ToTimestampedDocTransform.prototype._transform = function _transform(temperature\
, encoding, callback) {
  this.push({when: Date.now(), temperature: temperature});
  callback();
};
```

Since now we no longer need to create a document, we can simplify the database writable stream:

db_write_stream.js:

```
var Writable = require('stream').Writable;
var util = require('util');

module.exports = DatabaseWriteStream;

function DatabaseWriteStream(options) {
  if (! (this instanceof DatabaseWriteStream))
    return new DatabaseWriteStream(options);
  if (! options) options = {};
  options.objectMode = true;
  Writable.call(this, options);
}

util.inherits(DatabaseWriteStream, Writable);

DatabaseWriteStream.prototype._write = function write(doc, encoding, callback) {
  insertIntoDatabase(doc, callback);
};

function insertIntoDatabase(doc, cb) {
  setTimeout(cb, 10);
}
```

We can finally instantiate and pipe these streams together:

client.js:

```
var DbWriteStream = require('./db_write_stream');
var db = DbWriteStream();

var JSONEncodeStream = require('./json_encode_stream');
var json = JSONEncodeStream();

var ToTimestampedDocumentStream = require('./to_timestamped_document_stream');
var doc = ToTimestampedDocumentStream();

var Thermometer = require('../thermometer');

var thermometer = Thermometer();

thermometer.pipe(doc).pipe(json).pipe(db);
```

Here we're taking advantage of the fact that `stream.pipe()` returns the target stream, which we can use to pipe into the next stream, and so on.

8. Pipe specialisation via feature detection

Piping streams can be more than just about the flow of data. It can also be about connecting two objects that, given the capability of detecting features, can adapt accordingly. Let's show some instances of that.

8.1 Analysing Mikeal/request

Mikeal Roger's `request` module is a popular and open-source module that you can use to make HTTP requests from a Node.js process. In the code for `request`, inside the request-building procedure, there's this piece of code:

```
self.on('pipe', function (src) {
  if (self.ntick && self._started) throw new Error("You cannot pipe to this stre\
am after the outbound request has started.")
  self.src = src
  if (isReadStream(src)) {
    if (!self.hasHeader('content-type')) self.setHeader('content-type', mime.loo\
kup(src.path))
  } else {
    if (src.headers) {
      for (var i in src.headers) {
        if (!self.hasHeader(i)) {
          self.setHeader(i, src.headers[i])
        }
      }
    }
    if (self._json && !self.hasHeader('content-type'))
      self.setHeader('content-type', 'application/json')
    if (src.method && !self.explicitMethod) {
      self.method = src.method
    }
  }
})
```

What is this part of the code doing? First, it's adding a listener to the `pipe` event. When you do

```
source.pipe(dest)
```

amongst other things, the pipe implementation emits this `pipe` event on the destination stream, passing in the source as the sole event parameter. This enables the destination of the pipe to be aware of when and what it's being piped to. This allows us to hook in a series of feature detections which, as in this case, can use duck typing to determine the type of source stream, and do more.

In this case, the `request` module is detecting two types of source stream.

First it's detecting whether the source is a file stream. If that's the case, it takes the opportunity to parse the file extension and set the `content-type` header accordingly. If we do something like the following:

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

var read = fs.createReadStream('/path/to/my/file.json');
var req = read.pipe(request('http://example.com/url'));
```

the HTTP request we're originating here will bear the `application/json` content-type header.

If the source stream is not a file stream, request then checks if the source stream has a `headers` property. This may happen when the source is a server-side HTTP request object, as in this example:

```
var request = require('request');
var server = require('http').createServer(handleRequest);

function handleRequest(req, res) {
  req.pipe(request('http://example.com/some/url')).pipe(res);
}
```

In this case, this means that we're proxying a request: we're piping a server HTTP request directly into an HTTP client request, and in this case we copy the request headers onto the destination request.

We also copy the source stream `method` property value, which enables proxied requests to inherit the same HTTP method: a server-side request using the `POST` method generates a client HTTP request with a `POST` method just by piping it.

The request module also has some code for detecting whether the `_json` property is present and truthy:

```
if (self._json && !self.hasHeader('content-type'))
  self.setHeader('content-type', 'application/json')
```

This happens when we're piping a request into another request like this:

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

var req1 = request({
  url: 'http://example.com/some/service',
  json: true
});

var req2 = request('http://example2.com/some/other/service');

req1.pipe(req2);
```

The first request is expecting a JSON response body. Not only will this response body then be piped into the second request, but also the content type header.

8.1.1 Wrapping .pipe()

A few lines ago I showed you this example of where we're piping a server request into a client request:

```
var request = require('request');
var server = require('http').createServer(handleRequest);

function handleRequest(req, res) {
  req.pipe(request('http://example.com/some/url')).pipe(res);
}
```

By what we have described so far, we can now understand how the first pipe is able to relay the request headers and method into the request. But what is the second pipe doing?

To begin with, a request is a duplex stream. The request body is a writable stream, and the response body is a readable stream, all combined into one object. What you're really doing here is piping the response of the client request into the server response. When doing that the response headers and response status code are being passed on into the HTTP server response.

What mechanism is enabling this? By reading the request module source code we can see the following part:

```
Request.prototype.pipe = function (dest, opts) {
  if (this.response) {
    if (this._destdata) {
      throw new Error("You cannot pipe after data has been emitted from the resp\
onse.")
    } else if (this._ended) {
      throw new Error("You cannot pipe after the response has been ended.")
    } else {
      stream.Stream.prototype.pipe.call(this, dest, opts)
      this.pipeDest(dest)
      return dest
    }
  } else {
    this.dests.push(dest)
    stream.Stream.prototype.pipe.call(this, dest, opts)
    return dest
  }
}
```

Here we can see that the request object is overriding pipe. If the request already has a response and the response body hasn't been processed yet, it's calling the `pipeDest` method on it. If no response has arrived yet, it's adding the destination stream to an internal list; then it proceeds with applying the default `pipe` implementation.

By reading the code we can also see that, once the response arrives, the request is also calling the `pipeDest` method for each destination that was collected in the overriden pipe:

```
Request.prototype.onResponse = function (response) {
  var self = this
  // ...
  self.dests.forEach(function (dest) {
    self.pipeDest(dest)
  })
  //...
}
```

This means that all the secret sauce is in a) trapping the destination streams by overriding `pipe` and b) calling `this.pipeDest` on each one once we get a response.

Let's then take a peek into the `pipeDest` method:

```
Request.prototype.pipeDest = function (dest) {
  var response = this.response
  // Called after the response is received
  if (dest.headers && !dest.headersSent) {
    if (hasHeader('content-type', response.headers)) {
      var ctname = hasHeader('content-type', response.headers)
      if (dest.setHeader) dest.setHeader(ctname, response.headers[ctname])
      else dest.headers[ctname] = response.headers[ctname]
    }

    if (hasHeader('content-length', response.headers)) {
      var clname = hasHeader('content-length', response.headers)
      if (dest.setHeader) dest.setHeader(clname, response.headers[clname])
      else dest.headers[clname] = response.headers[clname]
    }
  }
  if (dest.setHeader && !dest.headersSent) {
    for (var i in response.headers) {
      dest.setHeader(i, response.headers[i])
    }
    dest.statusCode = response.statusCode
  }
  // ...
}
```

We can then see that the `pipeDest` method is, under certain conditions, copying the response headers and status code into the destination. This is why the second pipe call in our HTTP server propagates the response headers.

9. Summary

In JavaScript, and specially Node.js, I/O is always asynchronous: it relies on providing functions to be invoked when a certain event occurs. Particulary for operations with one definitive result, the callback pattern is a convention frequently used to notify of results or failures of a given operation.

You can use the async library as a pocket knife of asynchronous flow control. You can use `async.series` to chain operations and `async.parallel` to handle operations performed in parallel. You can use `async.waterfall` to avoid keeping global state between serial calls.

You can also asynchronously iterate over collections using `async.each`, `async.eachSeries`. You can use `async.eachLimit` to limit the maximum amount of pending parallel operations.

To tightly control the usage of a given resource you can place it behind a queue by using `async.queue`.

If you need to observe state changes throughout time, an Event Emitter is a pattern that allows you to decouple an event producer from the event consumers.

If you are transmitting similar data from a source into a destination, you can use streams to abstract the data source (a Readable Stream) and the data destination (a Writable Stream). You can also use Transform Streams if you need to filter or modify that data in some way.

Besides just chaining streams, a writable stream can listen to the `pipe` event and detect the features of a readable stream, using that to its advantage. A readable stream can also detect and use certain features of a writable stream, by wrapping the `pipe` method.