

Prerequisites

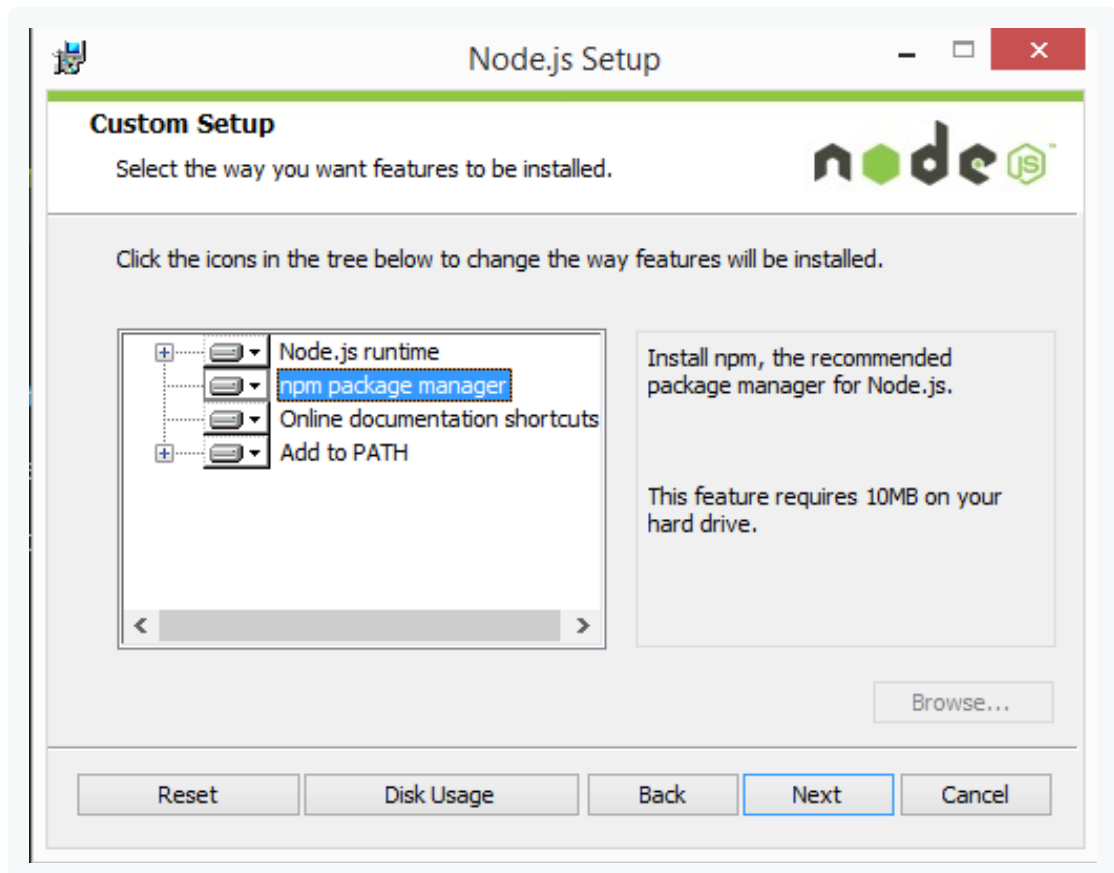
Node isn't a program that you simply launch like Word or Photoshop: you won't find it pinned to the taskbar or in your list of Apps. To use Node you must type command-line instructions, so you need to be comfortable with (or at least know how to start) a command-line tool like the Windows Command Prompt, PowerShell, [Cygwin](#), or the Git shell (which is installed along with [Github for Windows](#)).

Installation Overview

Installing Node and NPM is pretty straightforward using the installer package available from the Node.js® web site.

Installation Steps

1. **Download the Windows installer from the [Nodes.js® web site](#).**
2. **Run the installer** (the .msi file you downloaded in the previous step.)
3. **Follow the prompts in the installer** (Accept the license agreement, click the NEXT button a bunch of times and accept the default installation settings).

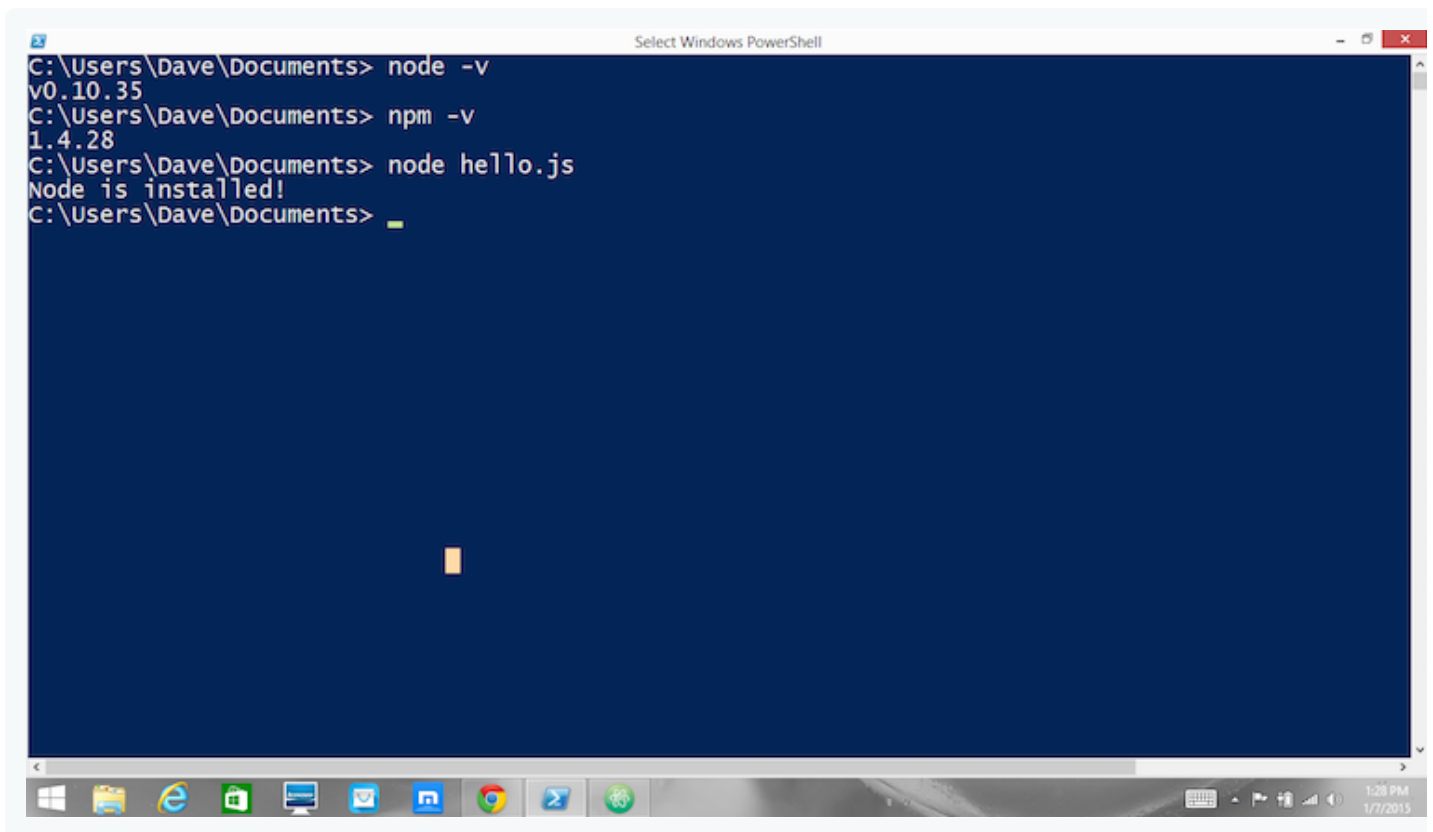


4. **Restart your computer.** You won't be able to run Node.js® until you restart your computer.

Test it!

Make sure you have Node and NPM installed by running simple commands to see what version of each is installed and to run a simple test program:

- **Test Node.** To see if Node is installed, open the Windows Command Prompt, Powershell or a similar command line tool, and type `node -v`. This should print a version number, so you'll see something like this `v0.10.35`.
- **Test NPM.** To see if NPM is installed, type `npm -v` in Terminal. This should print NPM's version number so you'll see something like this `1.4.28`
- **Create a test file and run it.** A simple way to test that node.js works is to create a JavaScript file: name it `hello.js`, and just add the code `console.log('Node is installed!');`. To run the code simply open your command line program, navigate to the folder where you save the file and type `node hello.js`. This will start Node and run the code in the `hello.js` file. You should see the output `Node is installed!`.



```
C:\Users\Dave\Documents> node -v
v0.10.35
C:\Users\Dave\Documents> npm -v
1.4.28
C:\Users\Dave\Documents> node hello.js
Node is installed!
C:\Users\Dave\Documents> _
```

The screenshot shows a Windows PowerShell window titled "Select Windows PowerShell". The terminal output shows the following commands and results:

- `node -v` returns `v0.10.35`
- `npm -v` returns `1.4.28`
- `node hello.js` returns `Node is installed!`

The Windows taskbar is visible at the bottom, showing icons for various applications and the system clock indicating 1:28 PM on 1/7/2015.

First things first: you should have node.js installed on your computer and available on your PATH. If you do not, please [download node.js now](#).

To ensure that you have node.js installed, open a command prompt and make sure the following commands exist and are at least the versions below:

```
$ node --version  
v0.10.13
```

```
$ npm --version  
1.3.2
```

Essentials for getting started

Node.js

Download and install the most up to date version of Node.JS:

<http://nodejs.org/>

MongoDB (for a later lab)

Download and install MongoDB:

Installers:

<http://www.mongodb.org/downloads>

Installation Instructions:

<http://docs.mongodb.org/manual/installation/>

A nice text editor

Sublime Text

A good text editor is essential for completing these labs. I recommend Sublime Text because it is cross-platform, free, and has a very simple to use interface:

<http://www.sublimetext.com/>

WebStorm

If you want a more powerful Integrated Development Environment try WebStorm:

<http://www.jetbrains.com/webstorm/>

The Node.js REPL

In this lab you will learn how to use the Node.js REPL (read-eval-print loop) to execute ad-hoc JavaScript statements. By the end of this lab you should be comfortable using the REPL and defining strings, objects, arrays, and functions in JavaScript.

- [Starting the Node.js REPL and executing statements](#)
- [Interacting with arrays](#)
- [Interacting with objects](#)
- [Creating and calling functions](#)
- [Multi-line statements in the REPL](#)
- [Exiting the Node.js REPL](#)

Starting the Node.js REPL and Executing Statements

To launch the Node.js REPL open a command prompt or terminal and execute `node`. Once open, evaluate a few simple expressions:

```
> 1 + 5
6
```

```
> var add = function (a, b) { return a + b; }
undefined
```

```
> add(1, 5)
6
```

Notice the result of the statement executed is printed on the following line without `>`. If you made a typo you can cancel your statement by pressing `CTRL+C` once.

If you forgot to assign the value of the previously executed statement, the Node.js REPL provides a useful syntax to access the previous result through `_`.

```
> "Node.js Rocks!"
'Node.js Rocks!'
```

```
> _
'Node.js Rocks!'
```

```
> var lastResult = _
undefined
```

```
> lastResult
'Node.js Rocks!'
```

Interacting with Arrays

A major difference between arrays in JavaScript and many other languages is that they are mutable and the size is not required upon creation. Use the array initializer (or array syntax) to create arrays:

```
> [1, 2]
[1, 2]

> [1,2].length
2
```


Adding an item to an array existing array:

```
> var a = ['apple', 'banana', 'kiwi']
undefined

> a.length
3

> a.push("lemon")
4 // `push` returns the size of the array after the push operation complete

> a.unshift("lime")
5 // `unshift` adds an element to the beginning of the array and returns th
```



Now inspect the contents of your array:

```
> a
[ 'lime',
  'apple',
  'banana',
  'kiwi',
  'lemon' ]
```

Removing an item from an array:

```
> a.pop()
'lemon' // `pop` removes and returns the last value in the array.

> a.shift()
'lime' // `shift` removes and returns the first value in the array.
```

The `slice` function can be used to copy a portion of an array to a new array. It does not modify the original array; rather, it copies it and returns a portion. It takes two arguments: a start index and end index. The end index is not inclusive.

```
> a
['apple', 'banana', 'kiwi']
```

```
> a.slice(0, 1)
['apple']

> a
['apple', 'banana', 'kiwi'] // the original array is not changed.

> a.slice(0)
['apple', 'banana', 'kiwi'] // copies the entire array.
```

You can even grab the last 2 values from an array using slice.

```
> a.slice(-2, a.length)
[ 'banana', 'kiwi' ]
```

Interacting with Objects

An object is not much more than a collection of keys and values. An object can be created using the object initializer (or object syntax). Properties can be set using the dot operator:

```
> var o = {}
undefined

> o.foo
undefined

> o.foo = 'bar'
'bar'

> o.foo.length
3
```

The array syntax (brackets) can allow you to create properties on objects that would otherwise be impossible to access using the dot syntax above. An interesting note here is that objects in JavaScript can have any value as keys, not just strings or numbers.

```
> o['foo^bar'] = 'things'
'things'

> o['foo^bar']
'things'

> o.foo^bar //This won't work because of the special character
ReferenceError: bar is not defined
    at repl:1:8
    at REPLServer.self.eval (repl.js:110:21)
    at Interface.<anonymous> (repl.js:239:12)
    at Interface.EventEmitter.emit (events.js:95:17)
    at Interface._onLine (readline.js:202:10)
    at Interface._line (readline.js:531:8)
    at Interface._ttyWrite (readline.js:760:14)
    at ReadStream.onkeypress (readline.js:99:10)
    at ReadStream.EventEmitter.emit (events.js:98:17)
```

```
at emitKey (readline.js:1095:12)
```

What if we wanted the keys of our objects to be functions? That's ok too! If you don't quite understand the function syntax yet don't worry that's coming up next.

```
> var x = function () { return 101; }
undefined
> var o = {}
undefined
> o[x] = 1
1
> o[x]
1
> x() //look it's just a function!
101
```

The array syntax `o['foo']` and the dot syntax `o.foo` can be used interchangeably for simple string values.

Objects can be composed of other objects:

```
> o.bar = [1, 2, 3, 4]
[1, 2, 3, 4]

> o.bar.length
4

> o.foobar = function () { return 'foo bar!'; }
[Function]

> o.foobar()
'foo bar!'

> o['foobar']()
'foo bar!'
```

Creating and Calling Functions

JavaScript functions are declared using the `function` keyword. This will create functions called `foo` and `bar` that do nothing:

```
> var foo = function () {} // this syntax is known as a `function expression`
undefined

> foo
[Function]

> function bar () {} // this syntax is known as a `function declaration`
undefined

> bar
[Function: bar]
```


Both function declarations and expressions define functions the same way. If you use a function declaration, it may be possible to call the function in your code in lines of code above where it is defined. This practice is not recommended.

It's recommended to always use function expressions unless you understand function hoisting ([further reading](#)).

Functions that do not have a name are anonymous. For example, `function () { }` is considered anonymous. These are commonly used as callback arguments to other functions.

```
// declare a function that takes a callback argument
> var foo = function (callback) {
...     callback();
... }
undefined

// passing an anonymous function as a callback
> foo(function () {
...     // callback function body
... });
undefined
```

Multi-line Statements in the REPL

You may have noticed that the Node.js REPL allows for multi-line statements to be executed. When a line cannot be processed as a complete JavaScript statement the Node.js REPL prompts for more input, as demonstrated when starting a function declaration above.

The `...` indicates that the Node.js REPL expects more input. `CTRL+C` can be used to cancel the multi-line statement if it was made in error.

Now, define a multi-line function and execute it:

```
> var boo = function () {
...     return "Hello World!";
... }
undefined

> boo()
'Hello World!'
```

Exiting the Node.js REPL

Exiting the Node.js REPL can be done by keyboard interrupt or exiting the process.

To exit by keyboard interrupt, press `CTRL+C` twice.

To instruct the Node.js process to exit:

```
> process.exit()
```

A Node.js Shell

In this lab we will put together a simple shell. We will interact with the file system and network while learning some useful features of JavaScript.

- Working with standard input
- Implementing a print working directory command
- Interacting with the filesystem: reading a directory
- Interacting with HTTP: downloading a file

Working with Standard Input

Commands will be provided to our shell through the process' standard input. By default, Node.js does not activate the standard input stream. The first thing we will do is enable standard input and echo back what we read.

Create a new file called `shell.js` in a brand new directory and add the following:

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

Before we go any further, let's experiment with what the above code does. To run the file:

```
$ node shell.js
```

Type anything and press enter. Notice that the input is line buffered (the data is sent in to Node.js every time you press enter). To shut down the process press `CTRL+C`.

The output of your program might look something like this:

```
foo  
<Buffer 61 73 64 0a>  
bar  
<Buffer 62 61 72 0a>
```

You can see that we're printing out a `Buffer` object. That's because the `input` variable passed in to our `function (input) { ... }` callback does not contain the string value of your input directly.

It's worth noting that, at this point, the buffer exists completely outside of the JavaScript

memory space (`Buffer` is an object from the C++ Node.js internals). Interacting with this buffer will move data from between the native (C++) and JavaScript boundary. For example, calling `input.toString()` will create a new JavaScript string containing the entire contents of the `Buffer`. An optional encoding can be specified as the first argument of the `toString` function (for example, `input.toString('utf8')`).

Since we're working with relatively short strings, let's go ahead and call `input.toString()` on the `Buffer` object. Here's what your program should look like now:

```
process.stdin.on('data', function (input) {  
  console.log(input.toString());  
});
```

Now starting up the shell and typing any value will result in the expected output ending with the new line character.

The next step is to trim and parse the input string. The commands in our simple shell will take the form:

```
command [args...]
```

We can use a handy regular expression to separate the arguments from the command: `/(\w+)(.*)/`. We will then parse our arguments by splitting on white space.

```
process.stdin.on('data', function (input) {  
  var matches = input.toString().match(/(\w+)(.*)/);  
  var command = matches[1].toLowerCase();  
  var args = matches[2].trim().split(/\s+/);  
});
```

A side note

The result of `'some string'.split(/\s+/)` is an array `['some', 'string']`.

This example could have been done with `'some string'.split(' ')`, except that would not account for other types of white space or multiple white space characters. For example:

```
'some  string'.split(' ') would result in ['some', '', 'string'].
```

Feel free to check out the result of the above code block by logging out the value of `command` and `args`. You may want to add a little more logic to make this resilient to malformed input. We will leave that exercise up to you.

Implementing a Print Working Directory Command

`pwd` is a simple program to print out the current working directory. Let's implement this in our shell.

```
var commands = {
  'pwd': function () {
    console.log(process.cwd());
  }
};

process.stdin.on('data', function (input) {
  var matches = input.toString().match(/(\w+)(.*)/);
  var command = matches[1].toLowerCase();
  var args = matches[2].trim().split(/\s+/);

  commands[command](args);
});
```

To clarify what's happening above, here's sample output of executing the regular expression at the Node.js REPL. The input is `cmd_name arg1 arg2`.

```
> var input = 'cmd_name arg1 arg2'
'cmd_name arg1 arg2'
> var matches = input.match(/(\w+)(.*)/)
> matches
[ 'cmd_name arg1 arg2',      // matches[0]
  'cmd_name',               // matches[1]
  ' arg1 arg2',             // matches[2]
  index: 0,                 // matches[3]
  input: 'cmd_name arg1 arg2' // matches[4]
```

We are accessing `matches[1]` because it's the first group (groups are specified with the parenthesis). If you are unfamiliar with regular expressions, a good source to learn more is at <http://regular-expressions.info>.

Now, give your shell a try!

Start up the shell with the `node` command and execute our one and only command:

```
$ node shell.js
pwd
/users/you/simple-shell/
```

Interacting with the file system: Reading a Directory

The command `ls [directory]` prints the contents of a directory. If the directory argument is not specified it will print the contents of the current working directory.

First, we will import the `fs` module at the top of the file. The `fs` module is the Node.js core module for file system operations.

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

To implement `ls`, add a new property to our commands object named 'ls':

```
var commands = {
  'pwd': function () {
    console.log(process.cwd());
  }, // <----- Don't forget this comma!
  'ls': function (args) { // New property added here.
    fs.readdir(args[0] || process.cwd(), function (err, entries) {
      entries.forEach(function (e) {
        console.log(e);
      });
    });
  }
};
```

Let's talk for a moment about `args[0] || process.cwd()`:

Unlike many other languages, JavaScript doesn't care if you access an index out of bounds of an array. If an element does not exist at the requested index, `undefined` will be returned. Since `undefined` is considered false, Using the `x || y` syntax will test the existence of `x` and if it is false (doesn't exist), it will evaluate to `y`. This is a common pattern for assigning a default value.

There are plenty of other commands available to access the file system. Feel free to [peruse the documentation](#) and implement any other commands that interest you.

Here's what your program should look like:

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

var commands = {
  'pwd': function () {
    console.log(process.cwd());
  },
  'ls': function (args) {
    fs.readdir(args[0] || process.cwd(), function (err, entries) {
      entries.forEach(function (e) {
        console.log(e);
      });
    });
  }
};

process.stdin.on('data', function (input) {
  var matches = input.toString().match(/(\w+)(.+)/);
  var command = matches[1].toLowerCase();
  var args = matches[2].trim().split(/\s+/);

  commands[command](args);
});
```

The commands object can get a bit hairy when we're nesting so many brackets deep. We can instead define our commands object like so:

```
var commands = {};  
  
commands['pwd'] = function () {  
  console.log(process.cwd());  
};  
  
commands['ls'] = function (args) {  
  fs.readdir(args[0] || process.cwd(), function (err, entries) {  
    entries.forEach(function (e) {  
      console.log(e);  
    });  
  });  
};
```

In this example we're defining a container object `commands` and assigning anonymous functions to properties on that object. The major difference from before is that the function definitions are not part of the object initializer. There's no difference in behavior between the two approaches.

Interacting with HTTP: Downloading a File

Similarly to the `fs` module, Node.js core contains a `http` module which can be used to act as a HTTP client or server. In the next lab you will be creating a HTTP server, but for now, we will focus on creating a simple `wget` command to download a file.

Import the `http` module up top next to where you imported `fs`:

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

Now, we will use `http.get(url, callback)` to make a HTTP GET request and download the results to the file system.

```
var commands = {  
  'pwd': ... , // omitted for brevity  
  'ls': ... , // omitted for brevity  
  'wget': function (args) {  
    var url = args[0];  
    var file = args[1] || 'download';  
  
    http.get(url, function (res) {  
      var writeStream = fs.createWriteStream(file);  
      res.pipe(writeStream);  
  
      res.on('end', function () {  
        console.log(url + ' downloaded to file \'' + file + '\');  
      });  
    });  
  }  
}
```

```
};
```

Now in order to test this try to run the following: (it's very important to include `http://`)

```
node shell.js wget http://google.com
```

The above command will download the contents of the home page of google to `download`.

Let's talk about what's happening in the callback provided to `http.get` callback.

First, we're creating a writable stream to our file system using `fs.createWriteStream`, named `file` (the second argument, or `'download'` by default). This will create or overwrite the file.

Next, let's talk about `res.pipe(writeStream);`. The `res` (response) object given to us isn't the full HTTP response. Since the response may be large, Node.js gives us the response as soon as it can be useful: after the HTTP handshake has occurred. We have access to the complete set of HTTP headers at this point, but do not have access to the data from the response.

The response data is streamed, much like the standard input stream we opened to read commands from the terminal. The `res` object emits `res.on('data')` and `res.on('end')` events, which can be directly piped out to our writable stream, saving chunks of data to the file system and then ending the write.

If you get stuck, check the [manual on the HTTP module](#).

Creating a HTTP Server

Now that you've had a little experience downloading a file using HTTP, let's use the `http` module in a different way: to create a simple HTTP server that responds to requests with a simple HTML template.

- Using the HTTP module to listen for incoming connections
- Serving static content
- Generating dynamic content with templates

Using the HTTP Module to Listen for Incoming Connections

Let's start by creating a new file named `server.js`.

Import the `http` module and use it to create a server:

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

var server = http.createServer(function (req, res) {
  // TODO
});
```

The callback function `function (req, res) {}` passed into the `createServer` call will be invoked with every new request to our server. This callback is passed two arguments, the request (a readable stream) and the response (a writable stream).

Our new server is not yet bound to any port, and therefore cannot accept incoming connections. In order to bind to a port, call the `server.listen(port)` function.

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

var server = http.createServer(function (req, res) {
  // TODO
});

server.listen(3000);
```

Let's boot up our server and take a look:

```
$ node server.js
```

Now, open your browser and navigate to <http://localhost:3000>. You will notice that your

browser seems to hang and will eventually timeout. This is because our server is not yet doing anything useful with the incoming connection.

Let's start by responding to all requests with a 200 status code. This is HTTP speak for "everything is OK."

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

var server = http.createServer(function (req, res) {
  res.statusCode = 200;
  res.end();
});

server.listen(3000);
```

Restart the server and visit <http://localhost:3000> once again. This time there should be a page with no content.

Now, let's add some data so we can be sure we're actually sending the response back.

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

var server = http.createServer(function (req, res) {
  res.statusCode = 200;
  res.write('Hello World!');
  res.end();
});

server.listen(3000);
```

Serving Static Content

Import `fs` to read a file off our file system and send it along to the web browser.

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

var server = http.createServer(function (req, res) {
  res.statusCode = 200;

  fs.readFile('index.html', function (err, data) {
    if (!err) {
      res.write(data.toString());
      res.end();
    }
  });
});

server.listen(3000);
```

Before we fire up the server, we need to create the `index.html` file:

```
<html>
  <head>
    <title>My Node.js server</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Let's give the server a run, and check out our work!

```
$ node server.js
```

Generating Dynamic Content with Templates

It's boring to serve content that doesn't change! Let's create a simple template engine to serve dynamic views.

```
var templateEngine = function (template, data) {
  var vars = template.match(/\{\w+\}/g);

  if (!vars) {
    return template;
  }

  var nonVars = template.split(/\{\w+\}/g);
  var output = '';

  for (var i = 0; i < nonVars.length; i++) {
    output += nonVars[i];

    if (i < vars.length) {
      var key = vars[i].replace(/\{\}/g, '');
      output += data[key];
    }
  }

  return output;
};
```

`templateEngine` takes a template string and a data object and searches for the pattern `{variableName}` and replaces matches with `data.variableName`. Feel free to copy/paste this code unless you want extra practice writing JavaScript and regular expressions.

Let's use this template engine to parse the content of our `index.html` file.

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

var templateEngine = function (template, data) {
```

```

var vars = template.match(/\{\w+\}/g);

if (!vars) {
  return template;
}

var nonVars = template.split(/\{\w+\}/g);
var output = '';

for (var i = 0; i < nonVars.length; i++) {
  output += nonVars[i];

  if (i < vars.length) {
    var key = vars[i].replace(/\{\}/g, '');
    output += data[key]
  }
}

return output;
};

var server = http.createServer(function (req, res) {
  res.statusCode = 200;
  res.setHeader("Content-Type", "text/html");

  fs.readFile('index.html', function (err, data) {
    if (!err) {
      res.write(templateEngine(data.toString(), {})); // use our template engine
      res.end();
    }
  });
});

server.listen(3000);

```

Now try this in the browser. You'll notice that the output is the same. Let's update `index.html` to take advantage of our template engine.

```

<html>
  <head>
    <title>My Node.js server</title>
  </head>
  <body>
    <h1>Hello {name}!</h1>
    <ul>
      <li>Node Version: {node}</li>
      <li>V8 Version: {v8}</li>
      <li>URL: {url}</li>
      <li>Time: {time}</li>
    </ul>
  </body>
</html>

```

The above modifications require several properties on our data object (name, node, v8, url, time). Let's assign those:

```
// ... code omitted for brevity
fs.readFile('index.html', function (err, data) {
  if (!err) {
    res.write(templateEngine(data.toString(), {
      name: 'Ryan Dahl',
      node: process.versions.node,
      v8: process.versions.v8,
      time: new Date(),
      url: req.url
    }));
    res.end();
  }
});
// ... code omitted for brevity
```

On Your Own

At this point nowhere have we written any logic that returns different data depending on the path of the request. As such, our simple HTTP server responds to every path the same. For example: <http://localhost:3000/path/to/file/one> and <http://localhost:3000/path/to/file/two>. Experiment with `req.url` to see if you can route requests based on the resource requested. Another useful module is `url`.

Here's an example usage of the `url` module:

```
> var url = require('url')
undefined
> url.parse('http://localhost:3000/path/to/file?q=1')
{ protocol: 'http:',
  slashes: true,
  auth: null,
  host: 'localhost:3000',
  port: '3000',
  hostname: 'localhost',
  hash: null,
  search: '?q=1',
  query: 'q=1',
  pathname: '/path/to/file',
  path: '/path/to/file?q=1',
  href: 'http://localhost:3000/path/to/file?q=1' }
```

Asynchronous patterns in Node.js

A common problem and solution

First, let's see the problem we're trying to avoid:

```
for (var i = 0; i < 100; i++) {  
  setTimeout(function () {  
    console.log(i);  
  }, 100);  
}  
console.log('You might think this gets printed last.')
```

Copy the code above into a new file that you will use for the rest of this lab. The points being illustrated require to be running from a .js file.

Notice the only value printed is 100, which is the terminal condition for this loop. This is because the callback for `setTimeout` doesn't actually get executed until the for loop has completed. Any other statements after the for loop will also be executed before any of the callbacks. This is, in part, because Node is single threaded. The event loop hasn't had a chance to invoke your callback. In any case, all of the callbacks are being fired with the same value of `i`.

A solution

How do we avoid this? We introduce a closure. In JavaScript a closures are achieved through the use of functions. By creating a closure to capture the value of `i` we're able to ensure we print the correct value for each iteration of the loop. When performing asynchronous operations in a loop be sure to capture variables you intend to use.

```
for (var i = 0; i < 100; i++) {  
  (function (i) {  
    setTimeout(function () {  
      console.log(i);  
    }, 100);  
  })(i); // The current value of i is captured by self executing function  
}  
console.log('this still gets printed first.');
```

A note about scope and closures

JavaScript, unlike many other languages, only has a function level scope. This means that variables declared within a for loop or if block are accessible outside of the brackets.

A better solution

A better way to do this would be to avoid creating a new function with each iteration of the loop like so:

```
var printNumberLater = function (i) {
  setTimeout(function () {
    console.log(i);
  }, 100);
};

for (var i = 0; i < 100; i++) {
  printNumberLater(i);
}

console.log('this still gets printed first.')
```

Asynchronous calls in parallel and then joining them

So how do we go about printing out something after the loops work has been done?

More often than not there's a need to invoke a set of tasks asynchronously and then start a new set of tasks upon completion:

```
var totalNumbersToPrint = 100;
var finishedCount = 0;

var printNumberLater = function (i) {
  setTimeout(function () {
    console.log(i);

    finishedCount++;

    if (finishedCount === totalNumbersToPrint) {
      console.log('this finally gets printed after all work is done!');
    }
  }, 100);
};

for (var i = 0; i < totalNumbersToPrint; i++) {
  printNumberLater(i);
}
```

In the example above, note that we keep track of how many times the `setTimeout` callback is invoked. Once the value matches the number of callbacks we're waiting on we can continue with printing out our message. This can get quite ugly the

more complex your application becomes. A very good library for dealing with this is [async](#). Here's an example of how async could be used to clean up the code above.

Start from a brand new directory and install the async module.

```
mkdir async-example
cd async-example
npm init /* use default settings */
npm install async --save

var async = require('async');
var workToBeDone = [];

var printNumberLater = function (i) {
  return function (callback) {
    setTimeout(function () {
      console.log(i);
      callback();
    }, 100);
  };
};

for (var i = 0; i < 100; i++) {
  workToBeDone.push(printNumberLater(i));
}

async.parallel(workToBeDone, function () {
  console.log('this still gets printed last');
});
```

Imagine the above where you had multiple things needing to happen in parallel. Remember how we kept around the `finishedCount` counter a few examples back? Keeping track of all the counters for more complicated examples could get extremely messy. Using `async` we're able to avoid these arbitrary counters and organize our code better. Another pattern to use is called `promises`.

Install the `promise-extended` npm package with the following command: `npm install promise-extended`. Then create a new file called `promises.js` with the following contents:

Promises

```
var p = require("promise-extended");
var Promise = p.Promise;

var operations = [];

var printNumberLater = function (i) {
  var prom = new Promise();

  setTimeout(function () {
```



```

        console.log(i);
        prom.callback();
    }, 100);

    return prom.promise(); //Return the newly created promise right away.
};

for (var i = 0; i < 100; i++)
{
    operations.push(printNumberLater(i));
}

p.when(operations).then(function (bios) {
    console.log('executed after all operations complete.');
```

What are they?

Promises are an abstraction that sit on top of asynchronous actions giving a concrete value to work with that represents a future value.

```

var Promise = require("promise-extended").Promise;

function asyncAction(){
    var p = new Promise();
    process.nextTick(function(){
        p.callback(1);
    });
    return p.promise();
}

asyncAction().then(function(res){
    console.log(res); //1;
});
```

In the above example we have the `function` `asyncAction` which returns a promises that will resolve on the next event loop with `1`.

Ok so what?

So where promises become a powerful in helping managing the callback madness is when managing multiple async actions in parallel or ensuring actions happen in a given order.

Example of managing execution order

```

var Promise = require("promise-extended").Promise;

var asyncCount = (function(){
    var currentCount = 0;
    return function(){
        var p = new Promise();
        process.nextTick(function(){
            p.callback(++currentCount);
        });
    };
});
```

```

        return p.promise();
    }
}());

asyncCount()
    .then(function(currCount){
        console.log(currCount); //1

        //we return a promise from in here which will cause then to wait for
        return asyncCount();
    })
    .then(function(currCount){
        console.log(currCount); //2

        //returning another promise!
        return asyncCount();
    })
    .then(function(count){
        //now we have the result from the last asyncCount call which is 3
        console.log(count); //3
    });

```

Example of managing multiple async actions at once. Make sure to install the `promise-extended` node module with the following command:

```
npm install promise-extended --save
```

```

var p = require("promise-extended"),
    Promise = p.Promise;

var asyncWait = function(timeout){
    var ret = new Promise();
    setTimeout(function(){
        ret.callback(timeout);
    }, timeout);
    return ret.promise();
}

```

```

//use the when method to wait for all of the passed in promises to resolve.
p.when(
    asyncWait(1000),
    asyncWait(900),
    asyncWait(800),
    asyncWait(700),
    asyncWait(600),
    asyncWait(500)
).then(function(res){
    console.log(res); //[1000,900,800,700,600,500];
});

```

In the above example `when` returns a promise that waits for all the passed in promises to resolve.

Error handling.

Error handling with promises is simplified greatly because you do not have to worry about errors unless your code cares about it directly.

```
var asyncCount = (function(){
  var currentCount = 0;
  return function(){
    var p = new Promise();
    process.nextTick(function(){
      p.callback(++currentCount);
    });
    return p.promise();
  }
})();

asyncCount()
  .then(function(count){
    console.log(count); //1
    return asyncCount();
  })
  .then(function(){
    throw new Error("Oops an error occurred");
  })
  .then(function(){
    return asyncCount();
  })
  .then(function(res){/*wont be called*/}, function (err){
    console.log(err.message); //Oops an error occurred ...
  })
```

Using and Creating Modules

In the last few labs, you used `require()` to import the `fs` and `http` core modules. Now let's see how `require` works and create a module of our own.

- [How do modules work?](#)
- [Create and require a module](#)
- [Require a JSON file](#)
- [Require a core module](#)
- [Require a module from the node_modules directory](#)

How do modules work?

In Node.js, modules work by exporting methods or values. If this were visualized in JavaScript it might be similar to this:

```
var exports = {};  
  
(function() {  
  var a = 10;  
  
  exports.foo = a * 10;  
})();  
  
console.log(exports.a); // undefined  
console.log(exports.foo); // 100
```

Notice the `(function () { /* module code here */ })();`. This provides a new scope for the module to prevent pollution of the global object. In JavaScript, this is known as a closure.

Modules in Node.js are executed within their own scope (therefore, the outer closure isn't necessary). If written into a Node.js module, the same code sample would be written as:

```
var a = 10;  
  
exports.foo = a * 10;
```

Before we get much further, it's important to note that the `exports` and `module.exports` objects in Node.js are equivalent. They are both available because one conforms to the [CommonJS](#) standard and the other to the [RequireJS](#) standard.

Create and require a module

First create a folder called `shapes`. Inside of `shapes`, create a file called `circle.js` with the following contents:

```
var PI = Math.PI;

exports.area = function (r) {
  return PI * r * r;
};

exports.circumference = function (r) {
  return 2 * PI * r;
};
```

The module `circle.js` has exported the functions `area()` and `circumference()`. To export an object, add to the special exports object. Variables local to the module will be private. In this example, the variable `PI` is private to `circle.js`.

Next, add another file to `shapes` called `circle_example.js` with the following contents:

```
var circle = require('./circle');
var radius = 10;
console.log('The area of a circle with radius ' + radius + ' is ' + circle.
```

Now, run the app by typing `node circle_example.js` from within the `shapes` directory. You should see the following:

```
The area of a circle with radius 10 is 314.159265359
```

Note the use of `./` in `require('./circle.js')` - if the module name starts with this Node will load the a file relative to the current working directory. Omitting the `./` would instruct Node to look in `node_modules` or within Nodes core modules.

On your own try to create a rectangle module with similar operations.

Require a JSON file

A JSON file can be included in a project by doing a `require('./filename.json')`. This is because `.json` files are parsed as JSON text files automatically.

Let's create an example of a JSON configuration file. To begin, create a a file called `config.json` with the following contents:

```
{
  "poll-frequency": 10,
```

```
"urls": ["test.com", "test.org"],  
"data": { "name": "test", "encoding": "utf8" }  
}
```

Now, let's open a Node REPL and see how the module system loads JSON files. Notice that what's returned by `require` is a JavaScript object that resembles the contents of `config.json`. Once again, it's important to remember the `./` prefix when requiring local files. If you omit the prefix Node will assume you're looking to load something from the `node_modules` directory or from Node's core modules.

```
> var config = require('./config')  
undefined  
> config.urls  
[ 'test.com', 'test.org' ]
```

Require a core module

Node.js ships with several modules to perform various system operations. The core modules are defined in Node.js's source in the lib folder.

Core modules are always preferentially loaded if their file name is passed to `require()`. For instance, `require('http')` will always return the built in HTTP module, even if there is a file or npm module by that name.

Require a module from the node_modules directory

Modules can be installed using the Node.js package manager `npm` to be included into your application.

For the next example, let's take a list of numbers and sort and filter them. To do this, include the underscore.js module. First we need to install underscore by executing the following command from a brand new directory.

```
$ npm init /* fill out the defaults */  
$ npm install underscore --save
```

Now create a file called `sort_example.js` and add the following:

```
var underscore = require('underscore');  
  
var listOfNumbers = [1,5,4,3,7,6];  
  
var sortedList = underscore.sortBy(listOfNumbers, function (n) { return n;  
console.log(sortedList);
```

After you've mastered how to use modules and create them you can learn to publish publicly [at the npm website](https://www.npmjs.com/).

Module caching

Node automatically caches required modules based on the full OS path to that module. This has a few side effects. First, each module will ever only be loaded once from a given directory. That means any modifications to the exported functions, objects, or internal state will be available to all users of that module in your application. Secondly, you can load different versions of the same module based upon where it's being loaded from. This can be very nice when some of your dependencies use a different version of `underscore` for example.

JavaScript must-knows

JavaScript is a language that many developers are familiar with but lack a deep understanding. The language lends itself to new programmers, in part, because of its simple syntax. Many fundamental concepts and features of the language can be overlooked with a simple procedural approach. The goal of this lab is to give you a better understanding of some core concepts and important language features of JavaScript.

What is `this`?

Have a look at this snippet and pay special attention to the use of the keyword `this`.

```
var obj = {
  doSomething: function () {
    this.a = "bob";

    function doAnotherThing () {
      console.log("Name: " + this.a);
    };

    console.log("Name: " + this.a);
    doAnotherThing();
  }
};

//What does this print?
obj.doSomething();
```

Those unfamiliar with how `this` is handled may think the call to `obj.doSomething` would output the following (assuming the snippet above is saved in a file named `this.js`):

```
> node this.js
Name: bob
Name: bob
```

This is a very common mistake. What is actually output is this:

```
> node this.js
Name: bob
Name: undefined
```

Here is another example and then we will explain what is going on.

```
function Adder (a) {
  this.a = a;
```



```
};

Adder.prototype.addAsync = function (b) {
  setTimeout(function () {
    console.log(this.a + b);
  }, 10);
};

var r = new Adder(5);
r.addAsync(10);
```

This example is supposed to add two numbers together after 10 milliseconds. What is the output of this example? Some may think it would print 15, the result of 5 + 10. The correct answer is `NaN`. Why? The answer lies in the value of `this`.

If there is anything you will walk away from after this lab I hope you no longer write bugs that involve `this`.

The value of `this` is the object that a function is defined on. Inner functions or function calls that are not a part of the object will have the default object set to `this`. The default object in browsers is `window` and in Node.js is `global`.

```
var obj = { a: "Example" };

var printer = function () {
  console.log(this);
  console.log(this.a);
};

obj.p = printer;

printer() // => undefined

obj.p() // => Example
```

Notice how the value of the call to `printer` that was not attached to an object was `undefined`. Once we set the property `p` to the printer function and invoke it you will see that `this` now refers to the object defined. When an object is created from a constructor function using the `new` keyword, a brand new object is set to `this`. For example:

```
var ctor = function (a) {
  this.a = a;
};

ctor.prototype.print = function () {
  console.log(this.a);
};

var o1 = new ctor('test1');
o1.print(); // => test1

var o2 = new ctor('test2');
o2.print(); // => test2
```

So, you have seen the default behavior of how JavaScript handles the `this` keyword. The value of `this` can be controlled in a function call via a few methods available on `Function.prototype`: `apply`, `call`, and `bind`. Each of these methods allow you to modify the value of `this` when the function is called. The first two options (`apply` and `call`) invoke the function immediately, whereas the third (`bind`) provides a new function with `this` bound. The value of `this` is the first argument to each of these methods. Here is an example:

```
var obj = { a: "Example" };

//Note this is not defined on obj
var printer = function () {
  console.log(this.a);
};

var newPrinter = printer.bind(obj); // Not immediately invoked

newPrinter();           // => "Example"
printer.apply(obj);     // => "Example"
printer.call(obj);      // => "Example"
```

The difference between `apply` and `call` is that `apply` allows you to invoke the function with the arguments as an array; `call` requires the parameters to be listed explicitly.

Var

The keyword `var` is used to define variables. Unfortunately, JavaScript does not require the use of this keyword when defining variables. Forgetting to leave off the `var` keyword can pollute the global object with unnecessary properties. It can also create innocent looking bugs. Have a look at the following example, What is the output?

```
function doStuff() {
  for (i = 0; i < 5; i++) {
    console.log(i);
  }
}

function example() {
  for (i = 0; i < 5; i++) {
    doStuff();
  }
}

example();
```

At first glance it looks like it would output the numbers 0 to 5 - 5 times. Sadly, it does not; instead it outputs the number 0 - 5 just once! What is the problem? It is the fact that this example omits the use of the `var` keyword. To fix this we must declare the loop control

variables within the function. Fixing the above example looks like this (note the use of var):

```
function doStuff() {
  var i;
  for (i = 0; i < 5; i++) {
    console.log(i);
  }
}

function example() {
  for (var i = 0; i < 5; i++) {
    doStuff();
  }
}

example();
```

Using JavaScripts strict mode will prevent you from defining properties on the global object. There are other advantages to using strict mode, but this one is my favorite. Strict mode is applied within an execution context instead of over the entire JavaScript VM. To enable strict mode simply include the string `"use strict";` at the top of an execution context. Here is an example of using strict mode just for a single function.

```
var strictFunction = function () {
  "use strict";
  x = 1; // => Throws error because of strict mode!
};

var notSoStrictFunction = function () {
  a = 1; // Still assigns a to the global object because "use strict" has
};

strictFunction(); //throws error because x has not been defined
notSoStrictFunction(); // No problem defining a on the global object
```



Use `===` over `==`

It's highly advised to always use the `===` and `!==` (read as strictly (not) equals) equality operators. The difference between `===` and `==` is that the former enforces that the types of the objects match, so a string can never equal a number. Whereas `==` will return true for `1 == "1"`.

```
// Consider taking input from an input field. This will be a string.
var input = "10";

// Compare the string "10" to the number 10. Because of == the number is cc
if (input == 10) {
  console.log(input * 5);
  console.log(input + 5);
}
```

```
}  
  
if (input === 10) {  
  console.log('this will not be reached');  
}  
  
if ((typeof input == typeof 10) && (input == 10)) {  
  console.log('this is essentially the same as ===');  
}
```



Streams

Streams are one of the most amazing features of Node. However, it's not a new thing. *nix systems use I/O streams to move data from one process to another.

Read Streams

The first stream we will discuss is a read stream. As you might expect this stream supports only reading. Let's take a look at an example.

Start by creating a file called `example.txt` and add whatever contents you like. Create another file called `readable.js` and add the following:

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

var rs = fs.createReadStream('./example.txt');

rs.on('data', function (data) {
  console.log(data.toString());
});

rs.on('end', function () {
  console.log('end');
});
```

Now that we have used a readable stream let's write our own.

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

function TeamStream () {
  Readable.call(this);
  var best_sports_programs = ['K-State Football', 'K-State Basketball', '
  var i = 0;
  this._read = function () {
    var program = best_sports_programs[i++];
    this.push(program || null);
  }
};

var rs = new TeamStream();
var received = 0;

rs.on('data', function (data) {
  received++;

  console.log(data.toString());
```

```

    if (received == 1) {
      console.log('pausing for some time');
      rs.pause();
      setTimeout(function () { rs.resume(); }, 3000);
    }
  });

  rs.on('end', function () {
    console.log('ended');
  });

```

We get to learn a few different things in this example. First, we create a new readable stream. The `TeamStream` function is inheriting the prototype from Node core's `ReadableStream` and in its constructor calls the `ReadableStream` constructor. We must call the constructor of the core implementation in order to perform the necessary initialization of our stream. We are required to implement the internal method `_read` which is called on every read request. In this example the first read request is issued whenever we subscribe to the `'data'` event. There are other ways to do this which we will cover next.

You will notice in this example that we are able to control the flow of data from a readable stream through `pause` and `resume`. This is incredibly important to allow your application to keep up with incoming data and only receive data when needed.

The above example reads data by the buffer. However, we have more fine grained control over how much data we receive through the `read` method. The next example will give you an idea of the series of events that occurs when reading from a stream that can provide data at various times. This is exactly the situation when dealing with network connections. Feel free to copy and paste this snippet. Study the output and understand the chain of events before moving on.

```

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

var log = (function () {
  var startTime = new Date().getTime();

  return function (msg) {
    var now = new Date().getTime();
    console.log('+' + Math.floor((now - startTime) / 1000) + ' ' + msg);
  };
})();

util.inherits(LetterStream, Readable);

function LetterStream () {
  Readable.call(this);
  var letters = ['aaaaaa', 'bbbbbb', 'cccccc', 'dddddd'];

  var i = 0;
  this._read = function () {

```

```

        log('Internal read called.');
```

```

    var letter = letters[i++];
    if (i > 1) {
        var that = this;
        setTimeout(function () {
            log('Pushing data async.');
```

```

            that.push(letter);
        }, 2000);
    }
    else
    {
        log('Pushing data.');
```

```

        this.push(letter || null);
    }
};

var rs = new LetterStream();

rs.on('end', function () {
    log('End');
});

rs.on('readable', function () {
    log('Stream is readable');
```

```

    var c;
    while (c = rs.read(2))
    {
        log(c.toString());
    }

    log('Reached end of readable data available');
});

log('Reached end of program. It\'s now alive because of async callbacks per
```



We've successfully read 2 bytes from our underlying stream with each iteration of the loop. Despite the number of calls to `read` we only invoke our internal function when new data is actually needed.

Write Stream

An example of where write streams are used are HTTP responses or standard output. Just like `ReadableStream`s, `WritableStream`s have a built in mechanism for rate limiting. These streams are aware of when the underlying buffer has been flushed and provide the caller with this information. This is very useful so that you don't see more than a client can handle. If the client isn't able to accept information fast enough it will buffer in memory. Let's have a look at a simple example of a commonly used write stream. Open a node REPL and input the following:

```
process.stdout.write('Hello World');
```

Notice the REPL printed out Hello World and the result of the method call `true`. The return value indicates whether or not the buffer was able to be flushed immediately. If the return value is false that means the data had to be buffered in memory and will be flushed later. True means that the data was immediately written and is not being buffered.

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

// By setting the high water mark to 0 we say that the underlying buffer mu
var ws = new WritableStream({ highWaterMark: 0});


ws._write = function (data, enc, cb) {
  // Unless otherwise specified in options, the string we write further dc
  // this file is converted to a buffer.
  console.log('Received', data);

  setTimeout(function () {
    process.stdout.write(data, enc, cb);
  }, 1000);
};

var result = ws.write('This should be buffered\n');

console.log('write return value', result);

//We can get notified when the underlying buffer has been flushed by using
result = ws.write('This will also be buffered\n', function () {
  console.log('We were notified about this buffer being flushed.');
```



```
});

console.log('write (2) return value', result);
```

Pipe streams together

Much praise is given to node by its community for providing an extremely simple interface for chaining streams together using `pipe`. You may already be familiar that *nix OS's provide the ability to pipe the standard output of one process into the standard input of another using a `|`. Let's see how we could take our standard input and direct it to our processes output stream.

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

We can test this by placing the above contents in a file called pipe.js and running the following shell command.

```
echo 'This is going to be piped out' | node pipe.js
```

How easy was that! If we didn't have `pipe` here's how we'd accomplish the same task. You can see pipe is taking care of a lot for us. What you do not see in this example is

that `pipe` is automatically handling backpressure.

```
process.stdin.on('data', function (data) {  
  process.stdout.write(data);  
});
```