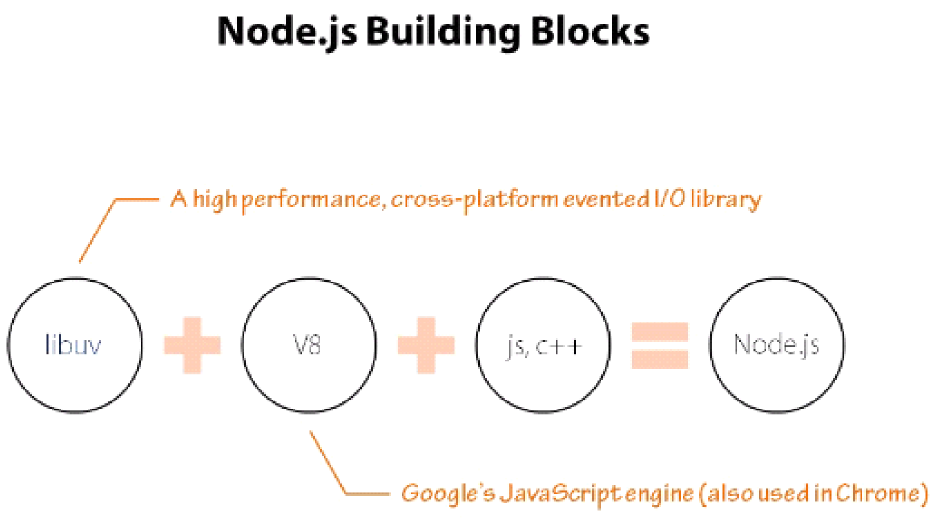
Introduction

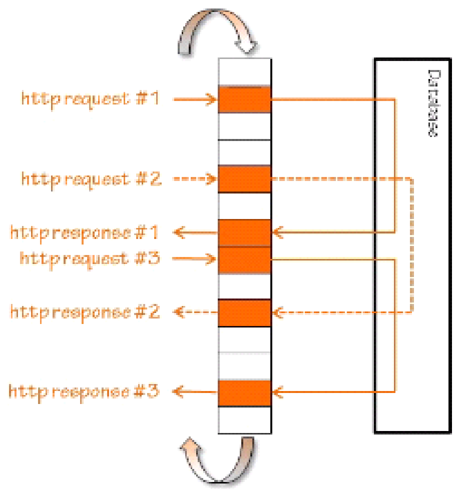
Wednesday, November 30, 2016
10:27 AM

Node.js is a server-side framework build on Google Crome's JS runtime.
Node.js is a application development platform that allows us to create standalone application development using JS.


Node.js Building Blocks
A high performance, cross-platform evented I/O library
libuv
V8
js, c++
Node.js
Google's JavaScript engine (also used in Chrome)

REPL -> Read Eval Print Loop

Event Loop



Using modules in your application

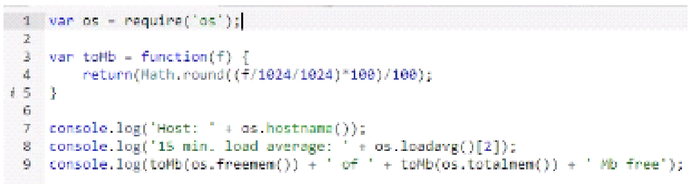
```
var foo = require('foo');
var Bar = require('bar');
var justOne = require('largeModule').justOne;

var f = 2 + foo.alpha;          ← Modules can export variables
var b = foo.beta() * 3;         ← ... including functions

var bar = new Bar();            ← Modules may export objects

console.log(justOne());
```

Three sources of Node modules

#1: Built-in Modules

- Come pre-packaged with Node
- Are require()'d with a simple string identifier
  - `var fs = require('fs');`

- A sample of built-in modules include:
  - fs
  - http
  - crypto
  - os



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

var toMb = function(f) {
    return(Math.round((f/1024/1024)*100)/100);
}

console.log('Host: ' + os.hostname());
console.log('15 min. load average: ' + os.loadavg()[2]);
console.log(toMb(os.freemem()) + ' of ' + toMb(os.totalmem()) + ' Mb free');
```

#2: Your Project's files

- Each .js file is its own module
- A great way to modularize your application's code
- Each file is require()'d with file system-like semantics:
  - `var data = require('./data');` ⟵ data.js in the same directory
  - `var foo = require('./other/foo');` ⟵ foo.js in the 'other' subdirectory
  - `var bar = require('../lib/bar');`

    bar.js in the 'lib' directory, "up and over" from this script's directory

Variables are marked for export via "module.exports"

one.js

```
var count = 2;

var doIt = function(i, callback) {
 // do something, invoke callback
}

module.exports.doIt = doIt;

module.exports.foo = 'bar';
```

two.js

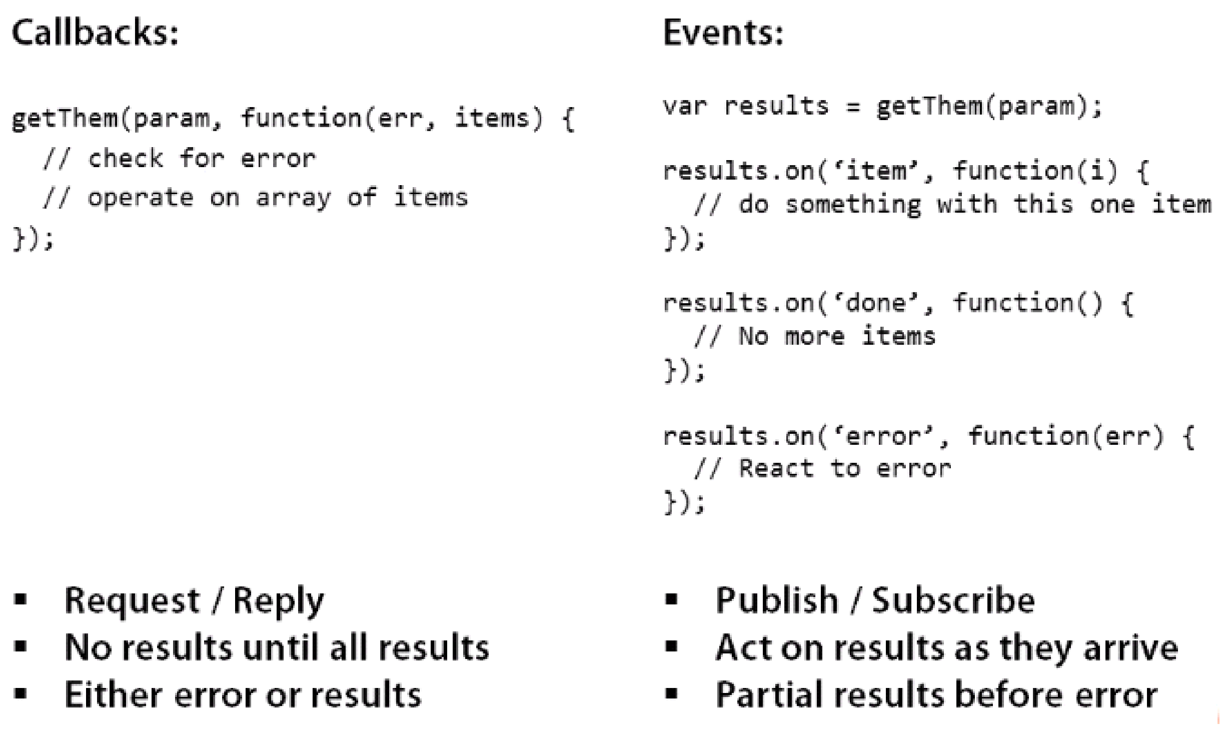
```
var one = require('./one');

one.doIt(23, function (err, result) {
  console.log(result);
});

console.log(one.foo);
```

⊗ ~~`console.log(one.count);`~~

#3: Third Party Modules via Node Package Manager (NPM) registry

- Installed via "npm install *module_name*" into "node_modules" folder
- Are require()'d via simple string identifiers, similar to built-ins
  - var request = require('request');
- Can require() individual files from within a module, but be careful!
  - var BlobResult = require('azure/lib/services/blob/models/blobresult');

- Some modules provide command line utilities as well
- Install these modules with "npm install –g *module_name*"
  - Examples include: express, mocha, azure-cli



Callbacks:

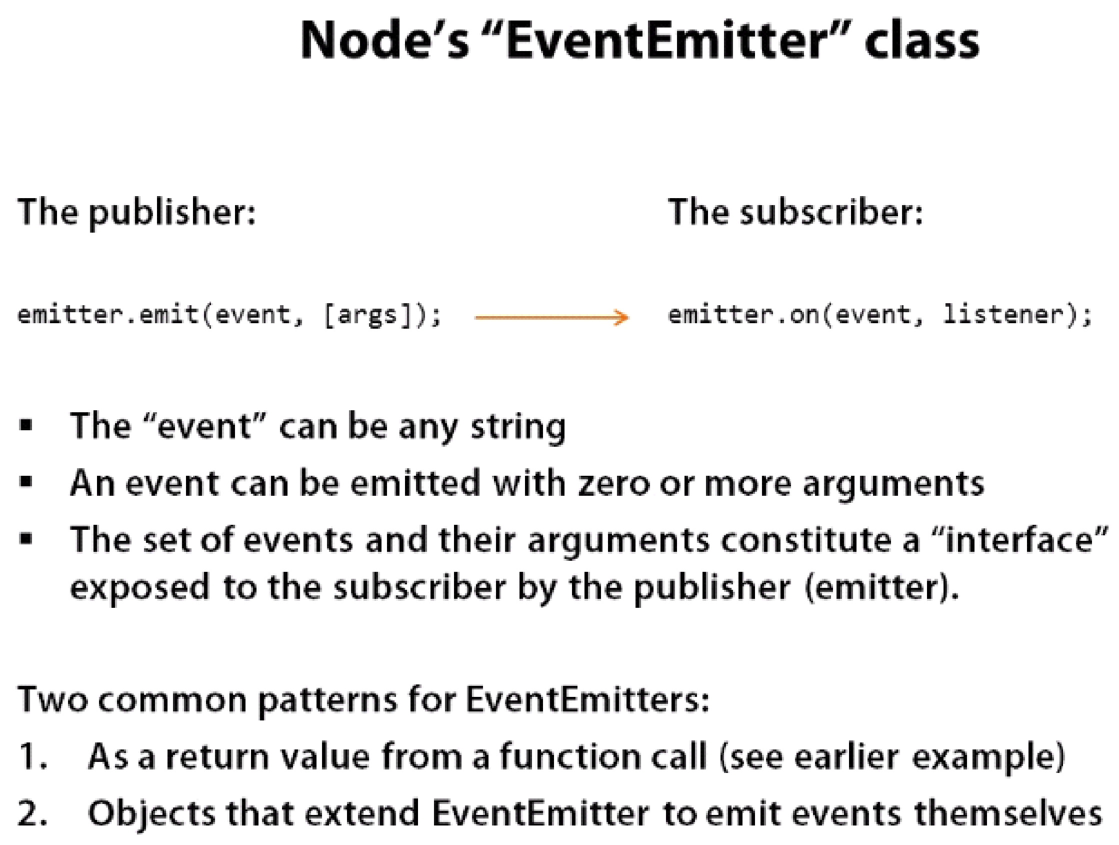
```
getThem(param, function(err, items) {
  // check for error
  // operate on array of items
});
```

- Request / Reply
- No results until all results
- Either error or results

Events:

```
var results = getThem(param);

results.on('item', function(i) {
  // do something with this one item
});

results.on('done', function() {
  // No more items
});

results.on('error', function(err) {
  // React to error
});
```

- Publish / Subscribe
- Act on results as they arrive
- Partial results before error



Node's "EventEmitter" class

The publisher:

```
emitter.emit(event, [args]);
```

$\longrightarrow$

The subscriber:

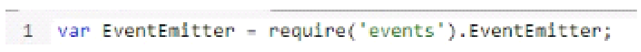
```
emitter.on(event, listener);
```

- The "event" can be any string
- An event can be emitted with zero or more arguments
- The set of events and their arguments constitute a "interface" exposed to the subscriber by the publisher (emitter).

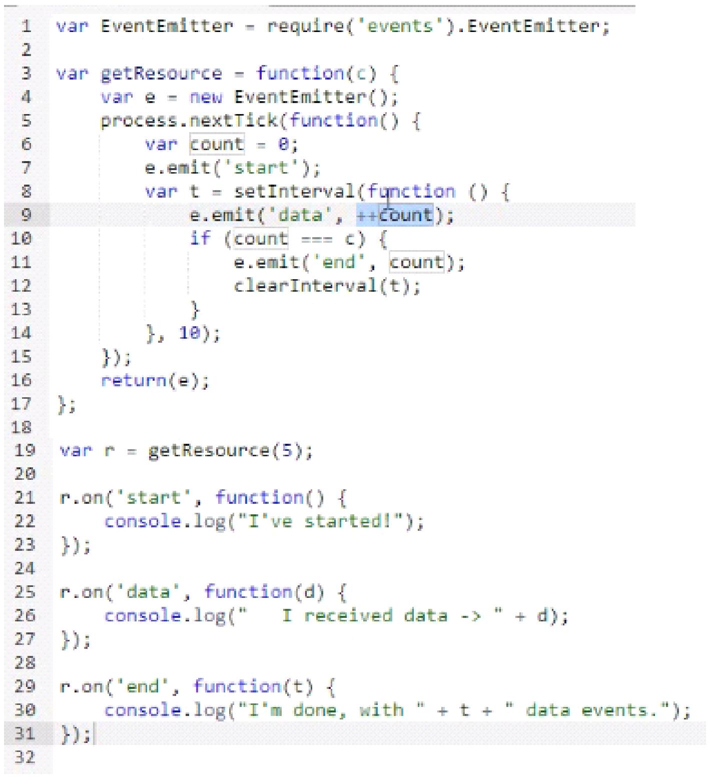
Two common patterns for EventEmitters:

1. As a return value from a function call (see earlier example)
2. Objects that extend EventEmitter to emit events themselves

First Pattern

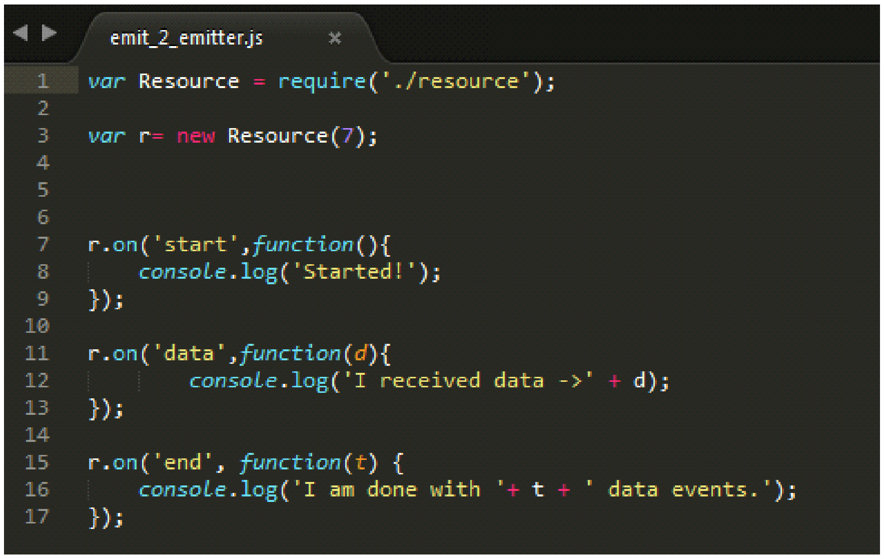


```
var EventEmitter = require('events').EventEmitter;
```



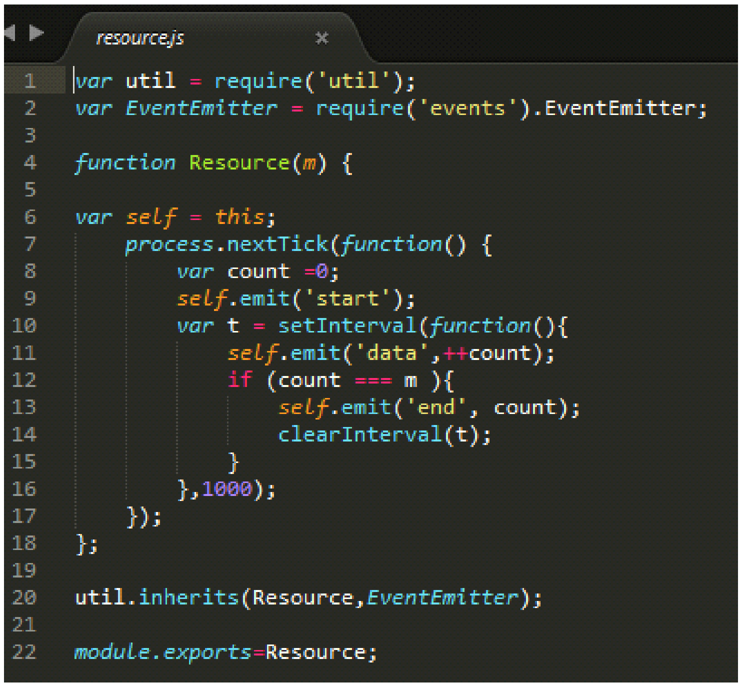
```
var EventEmitter = require('events').EventEmitter;

var getResource = function(c) {
    var e = new EventEmitter();
    process.nextTick(function() {
        var count = 0;
        e.emit('start');
        var t = setInterval(function () {
            e.emit('data', ++count);
            if (count === c) {
                e.emit('end', count);
                clearInterval(t);
            }
        }, 10);
    });
    return(e);
};

var r = getResource(5);

r.on('start', function() {
    console.log("I've started!");
});

r.on('data', function(d) {
    console.log("    I received data -> " + d);
});

r.on('end', function(t) {
    console.log("I'm done, with " + t + " data events.");
});
```

Second Pattern



emit_2_emitter.js

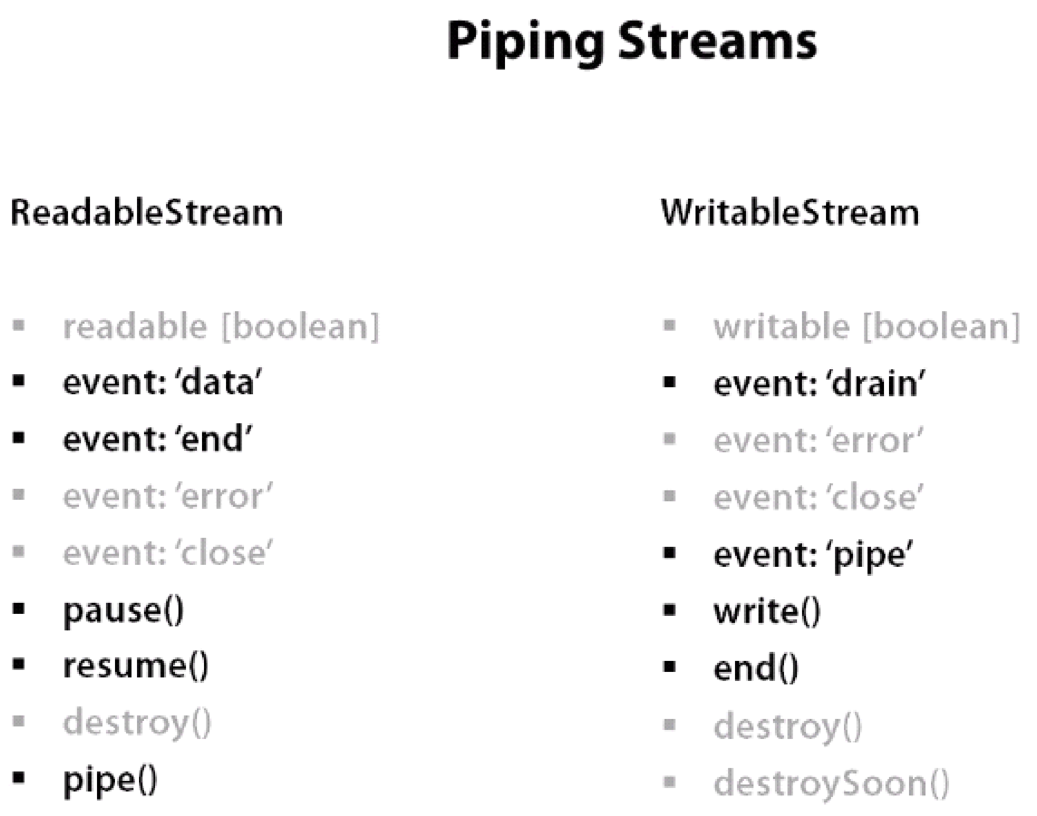
```
var Resource = require('./resource');

var r= new Resource(7);



r.on('start',function(){
    console.log('Started!');
});

r.on('data',function(d){
        console.log('I received data ->' + d);
});

r.on('end', function(t) {
    console.log('I am done with '+ t + ' data events.');
});
```



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

function Resource(m) {

var self = this;
    process.nextTick(function() {
        var count =0;
        self.emit('start');
        var t = setInterval(function(){
            self.emit('data',++count);
            if (count === m ){
                self.emit('end', count);
                clearInterval(t);
            }
        },1000);
    });
};

util.inherits(Resource,EventEmitter);

module.exports=Resource;
```

Streams in Node.js

- Streams are instances of (and extensions to) EventEmitter with an agreed upon "interface"
- A unified abstraction for managing data flow, including:
  - Network traffic (http requests & responses, tcp sockets)
  - File I/O
  - stdin / stdout / stderr
  - … and more!
- A stream is an instance of either
  - ReadableStream
  - WritableStream
  - … or both!
- A ReadableStream can be pipe()'d to a WritableStream
  - Applies "backpressure"



Piping Streams

ReadableStream	WritableStream
▪ readable [boolean]	▪ writable [boolean]
▪ event: ‘data’	▪ event: ‘drain’
▪ event: ‘end’	▪ event: ‘error’
▪ event: ‘error’	▪ event: ‘close’
▪ event: ‘close’	▪ event: ‘pipe’
▪ pause()	▪ write()
▪ resume()	▪ end()
▪ destroy()	▪ destroy()
▪ pipe()	▪ destroySoon()

When you invoke the **pipe()** function from the Readable Stream, you pass as a parameter the Writeable Stream you want to pipe to. This return emits the **event:pipe** in Writeable stream. The pipe function then begin an orchestration of events and functions between the two streams.

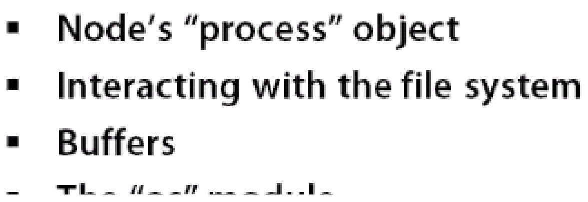
When a data arrives to the Readable Stream the **event:data** is emitted and the **write** function on the writeable stream is invoked with this data.

If at some point the **write** function returns a false value indicating that no more data should be written, the **pause** function from Readable Stream is called 'to stop the flow of the data'.

Then, once the writeStream is ready(**resume**) to receive more data, the **drain** event is emitted.

Once the readable finishes, the **end event** is emitted and the **end function** is called.

Assessing Local system



- Node’s “process” object
- Interacting with the file system
- Buffers
- The “os” module

- **Node's "process" object**
- **Interacting with the file system**
- **Buffers**
- **The "os" module**

The "process" object

- **A collection of Streams**
  - process.stdin
  - process.stdout
  - process.stderr
- **Attributes of the current process**
  - process.env
  - process.argv
  - process.pid
  - process.title
  - process.uptime()
  - process.memoryUsage()
  - process.cwd()
- **Process-related actions**
  - process.abort()
  - process.chdir()
  - process.kill()
  - process.setgid()
  - process.setuid()
  - … etc.
- **An instance of EventEmitter**
  - event: 'exit'
  - event: 'uncaughtException'
  - POSIX signal events ('SIGINT', etc.)

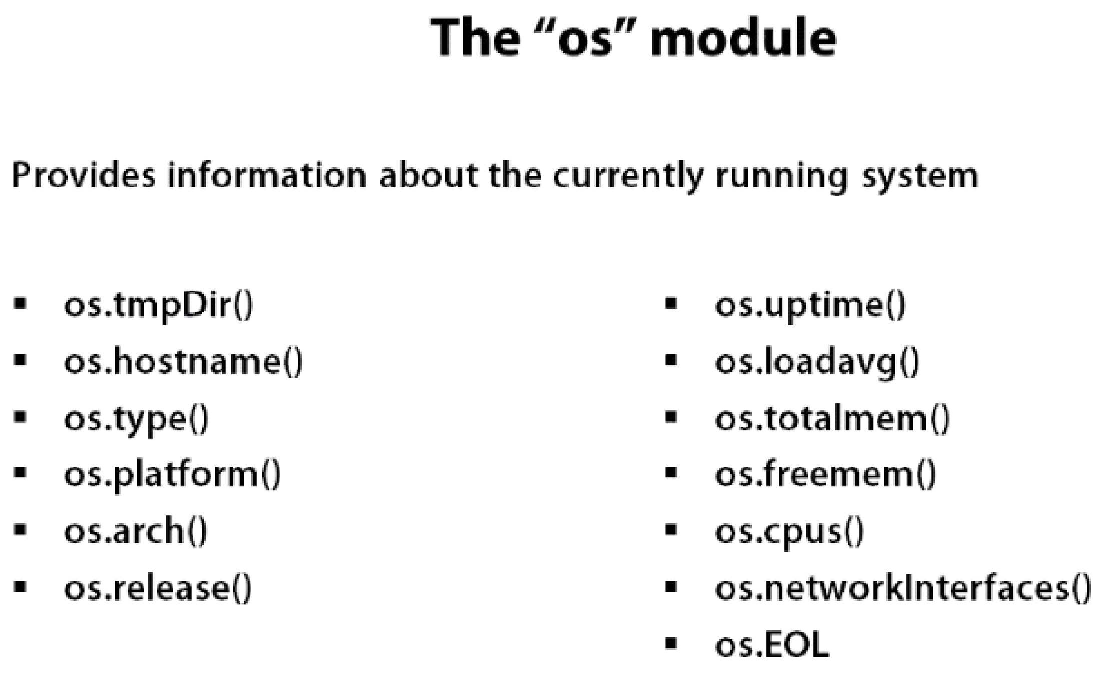
Interacting with the File System

- Wrappers around POSIX functions (both async and sync versions)
  - Functions include:

    rename, truncate, chown, fchown, lchown, chmod, fchmod, lchmod, stat, fstat, lstat, link, symlink, readlink, realpath, unlink, rmdir, mkdir, readdir, close, open, utimes, futimes, fsync, write, read, readFile, writeFile, and appendFile
  - For example: `fs.readdir(path, callback)` and `fs.readdirSync(path)`
- Stream oriented functions
  - `fs.createReadStream()` – returns an fs.ReadStream (a ReadableStream)
  - `fs.createWriteStream()` – returns an fs.WriteStream (a WritableStream)
- Watch a file or directory for changes
  - `fs.watch()` – returns an fs.FSWatcher (an EventEmitter)
  - 'change' event: the type of change and the filename that changed
  - 'error' event: emitted when an error occurs

What is a Buffer?

- JavaScript has difficulty dealing with binary data
- However, networking and the file system require it
- The Buffer class provides a raw memory allocation for dealing with binary data directly
- Buffers can be converted to/from strings by providing an encoding:
  - ascii, utf8 (default), utf16le, ucs2, base64, binary, hex
- Provides a handy way to convert strings to/from base64

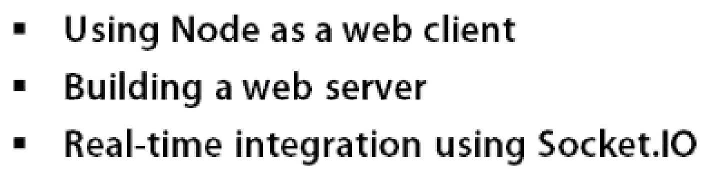


The “os” module

Provides information about the currently running system

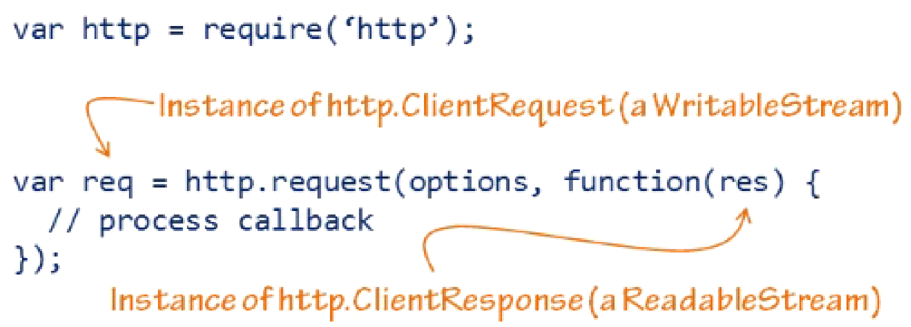
- os.tmpDir()
- os.hostname()
- os.type()
- os.platform()
- os.arch()
- os.release()
- os.uptime()
- os.loadavg()
- os.totalmem()
- os.freemem()
- os.cpus()
- os.networkInterfaces()
- os.EOL

Interacting with Web



- Using Node as a web client
- Building a web server
- Real-time integration using Socket.IO

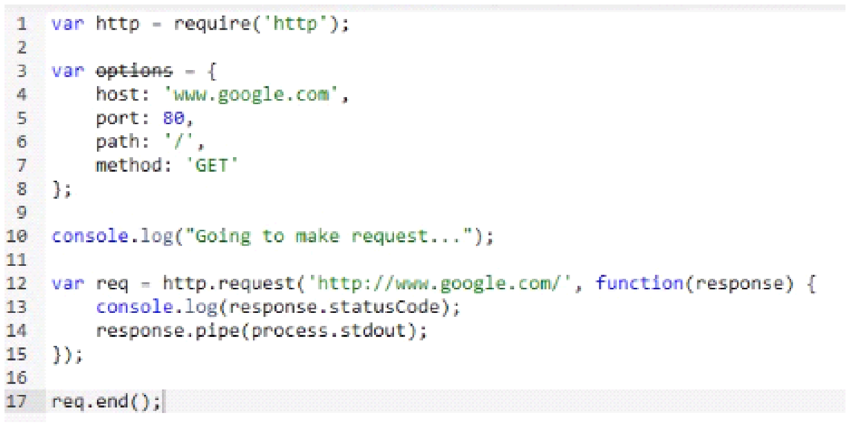
Making web requests in Node



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

var req = http.request(options, function(res) {
  // process callback
});
```

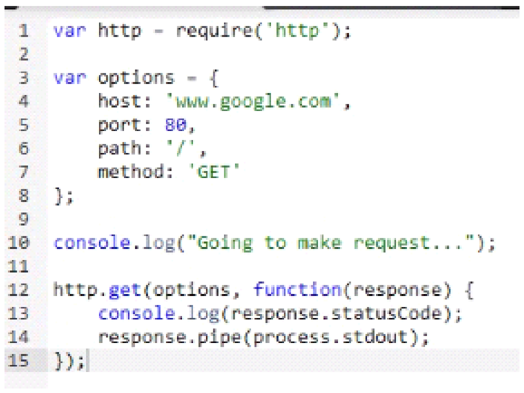
Instance of http.ClientRequest (a WritableStream)
Instance of http.ClientResponse (a ReadableStream)

- "options" can be one of the following:
  - A URL string
  - An object specifying values for host, port, method, path, headers, auth, etc.
- The returned ClientRequest can be written/piped to for POST requests
- The ClientResponse object is provided via either callback (shown above) or as a "response" event on the request object.
- http.get() available as a simplified interface for GET requests

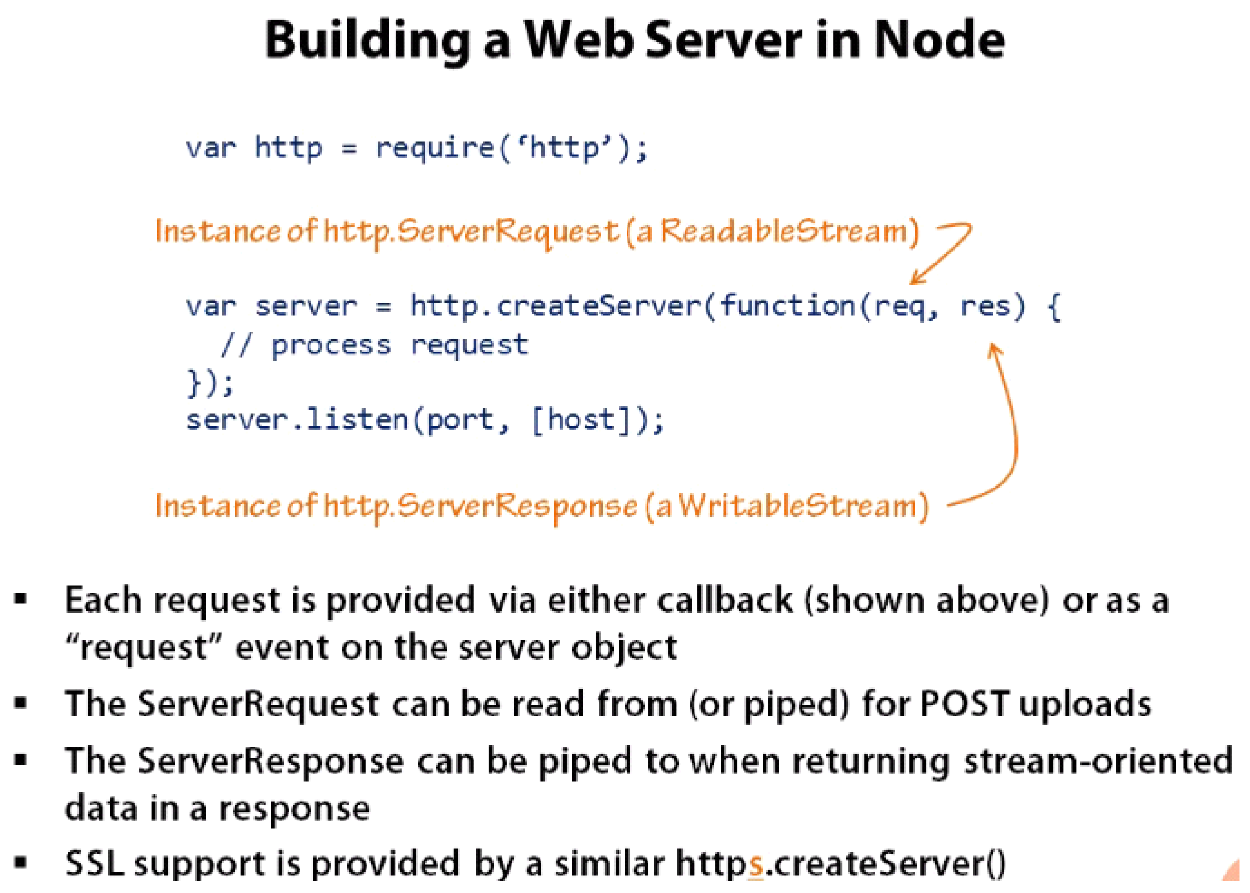


```
var http = require('http');

var options = {
    host: 'www.google.com',
    port: 80,
    path: '/',
    method: 'GET'
};

console.log("Going to make request...");

var req = http.request('http://www.google.com/', function(response) {
    console.log(response.statusCode);
    response.pipe(process.stdout);
});

req.end();
```

OR



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

var options = {
    host: 'www.google.com',
    port: 80,
    path: '/',
    method: 'GET'
};

console.log("Going to make request...");

http.get(options, function(response) {
    console.log(response.statusCode);
    response.pipe(process.stdout);
});
```



Building a Web Server in Node

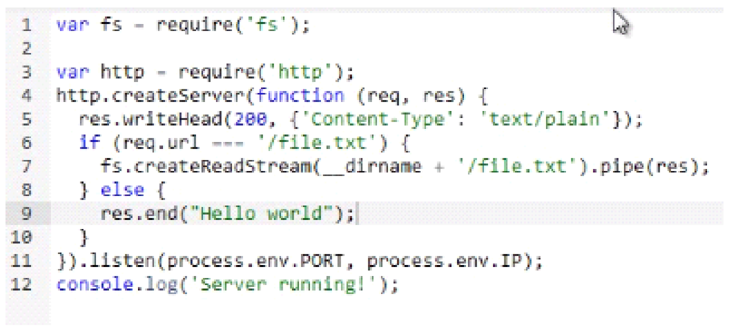
```
var http = require('http');
```

Instance of http.ServerRequest (a ReadableStream)

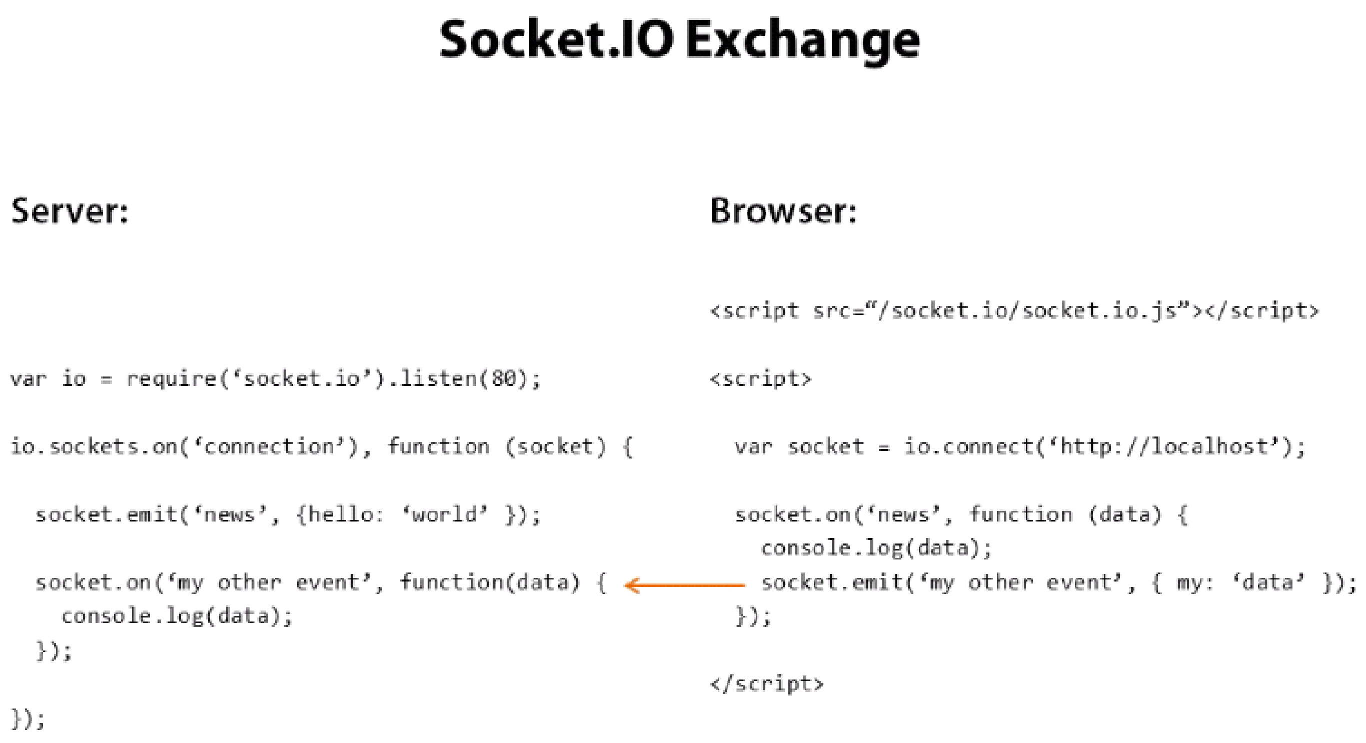
```
var server = http.createServer(function(req, res) {
  // process request
});
server.listen(port, [host]);
```

Instance of http.ServerResponse (a WritableStream)

- Each request is provided via either callback (shown above) or as a "request" event on the server object
- The ServerRequest can be read from (or piped) for POST uploads
- The ServerResponse can be piped to when returning stream-oriented data in a response
- SSL support is provided by a similar http<u>s</u>.createServer()



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

var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  if (req.url === '/file.txt') {
    fs.createReadStream(__dirname + '/file.txt').pipe(res);
  } else {
    res.end("Hello world");
  }
}).listen(process.env.PORT, process.env.IP);
console.log('Server running!');
```



Socket.IO Exchange

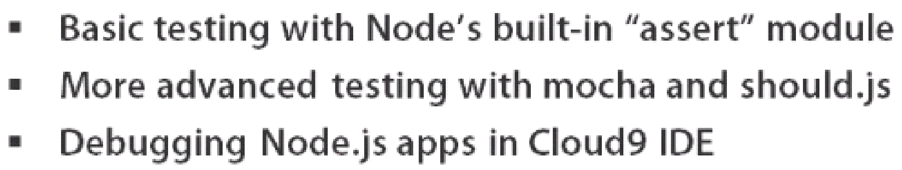
Server:

```
var io = require('socket.io').listen(80);

io.sockets.on('connection'), function (socket) {

  socket.emit('news', {hello: 'world' });

  socket.on('my other event', function(data) {
    console.log(data);
  });

});
```

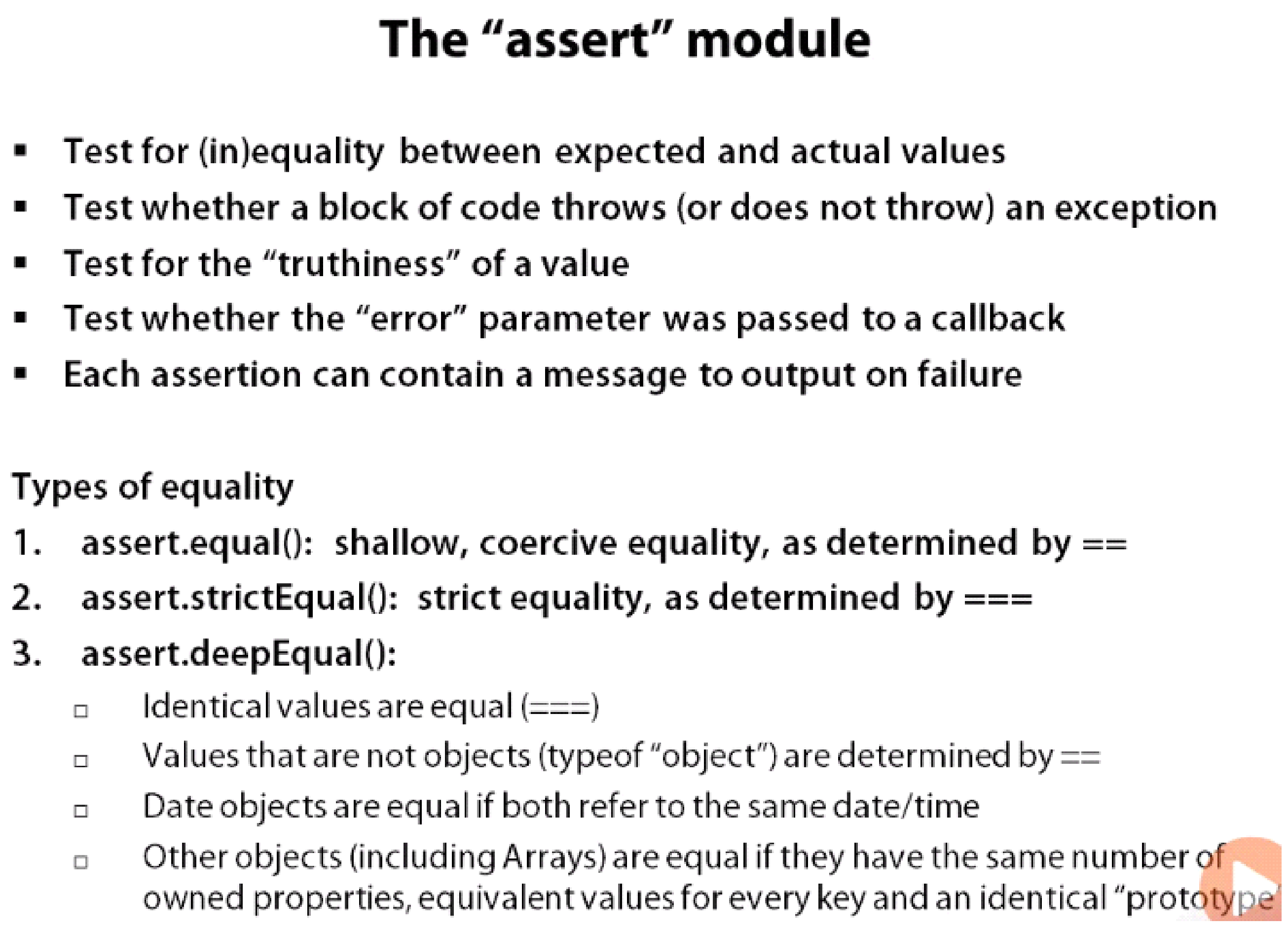
Browser:

```
<script src="/socket.io/socket.io.js"></script>

<script>

  var socket = io.connect('http://localhost');

  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'data' });
  });

</script>
```

Testing and Debugging



- Basic testing with Node's built-in "assert" module
- More advanced testing with mocha and should.js
- Debugging Node.js apps in Cloud9 IDE

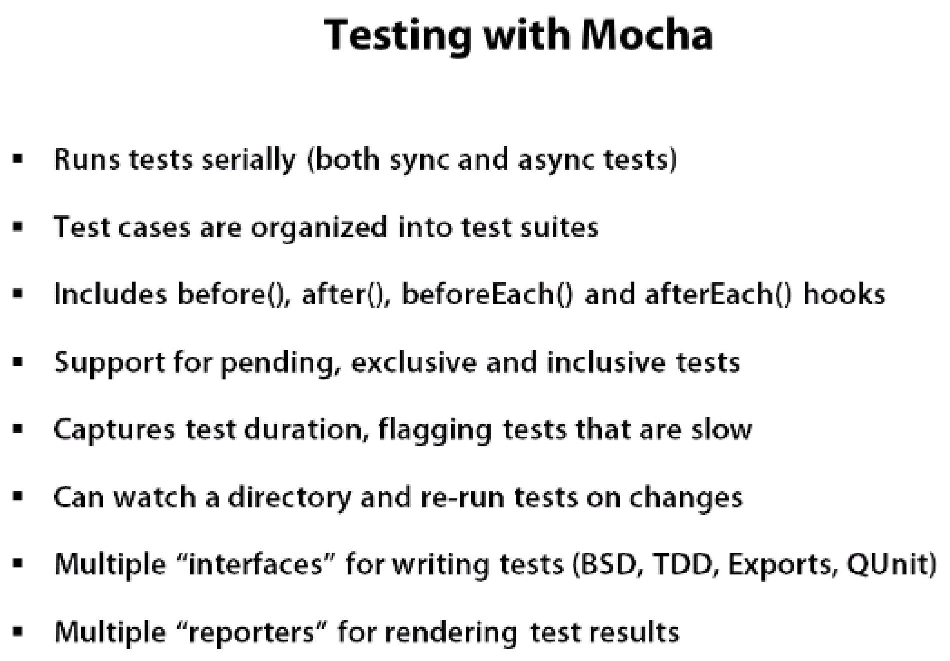


The “assert” module

- Test for (in)equality between expected and actual values
- Test whether a block of code throws (or does not throw) an exception
- Test for the “truthiness” of a value
- Test whether the “error” parameter was passed to a callback
- Each assertion can contain a message to output on failure

Types of equality

1. assert.equal(): shallow, coercive equality, as determined by ==
2. assert.strictEqual(): strict equality, as determined by ===
3. assert.deepEqual():
   - Identical values are equal (===)
   - Values that are not objects (typeof “object”) are determined by ==
   - Date objects are equal if both refer to the same date/time
   - Other objects (including Arrays) are equal if they have the same number of owned properties, equivalent values for every key and an identical “prototype

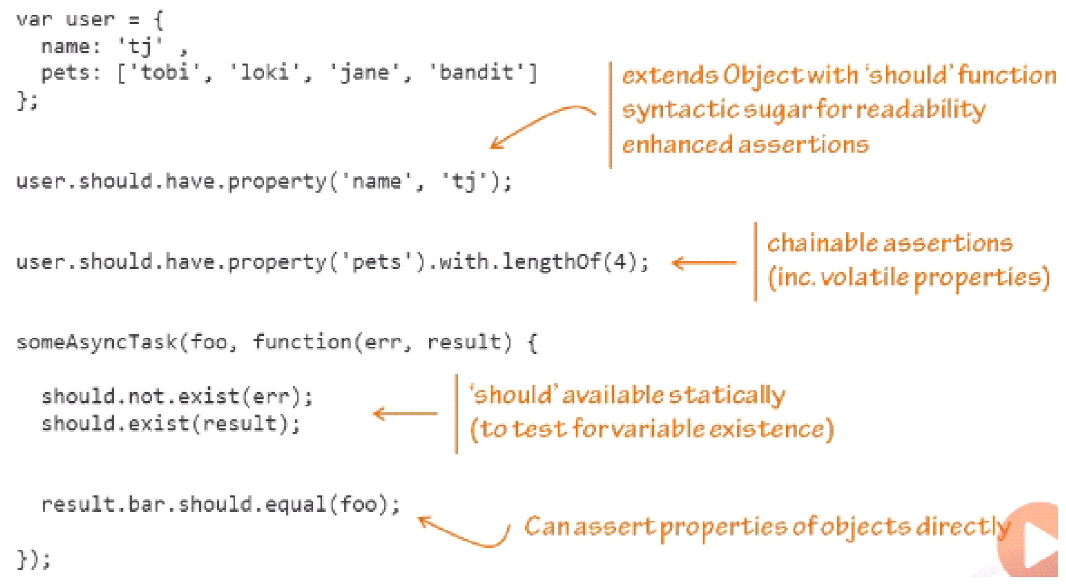


Testing with Mocha

- Runs tests serially (both sync and async tests)
- Test cases are organized into test suites
- Includes before(), after(), beforeEach() and afterEach() hooks
- Support for pending, exclusive and inclusive tests
- Captures test duration, flagging tests that are slow
- Can watch a directory and re-run tests on changes
- Multiple “interfaces” for writing tests (BSD, TDD, Exports, QUnit)
- Multiple “reporters” for rendering test results

Asserting with should.js

Extends Node's "assert" module with BDD style assertions



```
var user = {
  name: 'tj' ,
  pets: ['tobi', 'loki', 'jane', 'bandit']
};

user.should.have.property('name', 'tj');

user.should.have.property('pets').with.lengthOf(4);

someAsyncTask(foo, function(err, result) {

  should.not.exist(err);
  should.exist(result);

  result.bar.should.equal(foo);

});
```

extends Object with 'should' function
syntactic sugar for readability
enhanced assertions

chainable assertions
(inc. volatile properties)

'should' available statically
(to test for variable existence)

Can assert properties of objects directly

Scaling your Node Application

- Creating child processes in Node
- Scaling your Node app with the "cluster" module

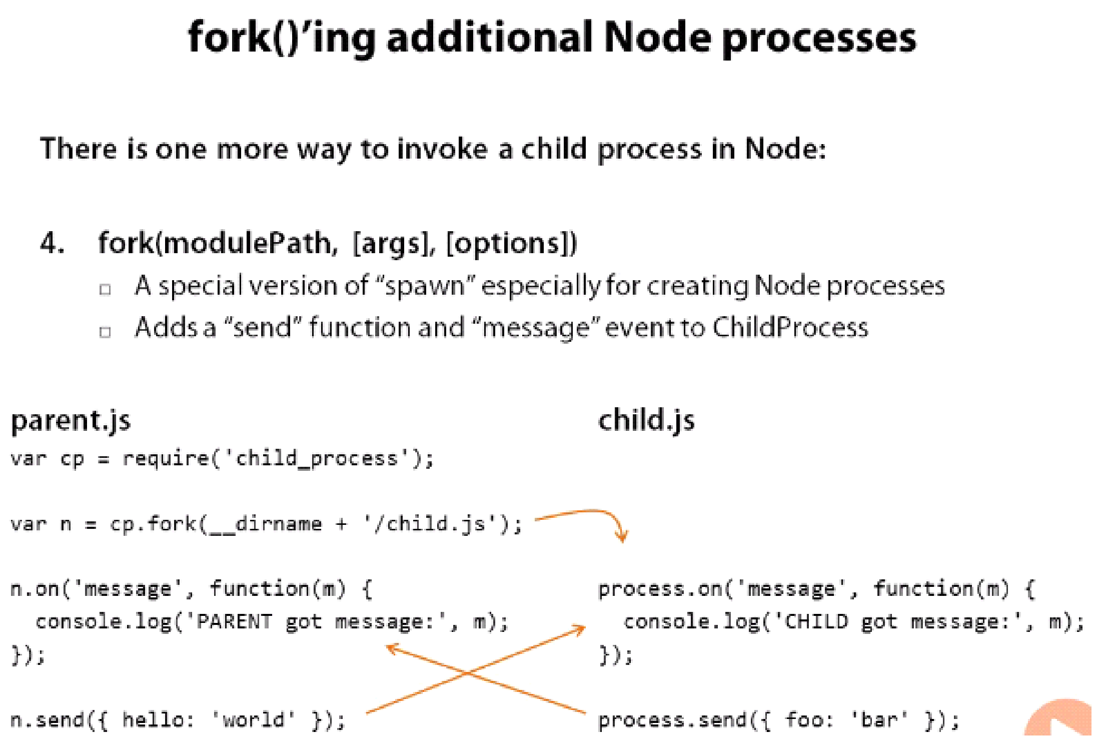
Creating Child Processes

The "child_process" module provides several ways to invoke a process:

1. spawn(command, [args], [options])
   - Launches a new process with "command" and "args"
   - Returns a ChildProcess object that…
     - is an EventEmitter and emits "exit", and "close" events
     - has streams for stdin, stdout and stderr that can be piped to/from

2. exec(command, [options], callback)
   - Runs "command" string in a shell
   - Callback is invoked on process completion with error, stdout, stderr

3. execFile(file, [args], [options], callback)
   - Similar to "exec", except "file" is executed directly, rather than in a subshell



fork()'ing additional Node processes

There is one more way to invoke a child process in Node:

4. fork(modulePath, [args], [options])
   - A special version of "spawn" especially for creating Node processes
   - Adds a "send" function and "message" event to ChildProcess

parent.js

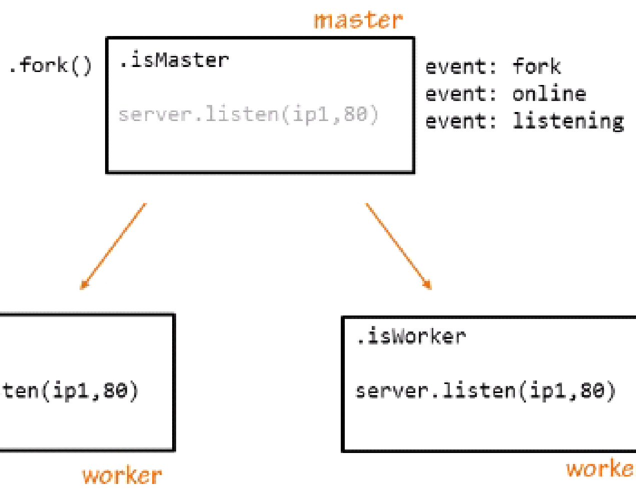
```
var cp = require('child_process');

var n = cp.fork(__dirname + '/child.js');

n.on('message', function(m) {
  console.log('PARENT got message:', m);
});

n.send({ hello: 'world' });
```

child.js

```
process.on('message', function(m) {
  console.log('CHILD got message:', m);
});

process.send({ foo: 'bar' });
```

Scaling with Node's "cluster" module

- An experimental module leveraging child_process.fork()
- Introduces a "Worker" class as well as master functions and events



Command prompt
- global
- process
- require
- module

Crome Console
- window
- document