

Node.js & You

Learn to Write the Realtime Web



Jim Snodgrass
jim@skookum.com
@snodgrass23

quick name intro

https://github.com/Skookum/sxsw13_nodejs_workshop

<http://skookum.github.com/WebChat/>

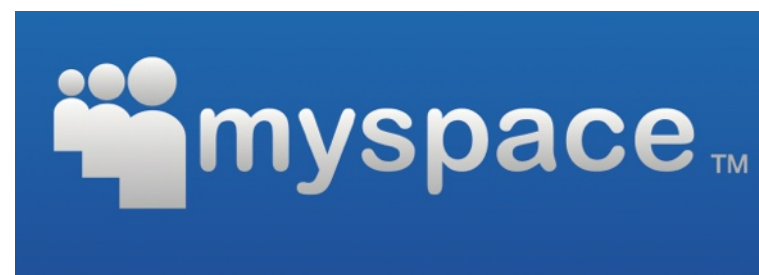
repo with examples and demos

audience survey

- * how many have laptops and will be participating?
- * how many know JS (more than JQuery?)
- * how many know any Node
- * how many have built something meaningful in Node
- * does anyone need help installing node or npm?



Who uses Node.js?





Why do I use Node.js?

I was first introduced to Node.js in one of Skookum's weekly tech talks by a coworker, Hunter Loftis, who had just began playing with it. This was on version 0.2.*. At first, this was a difficult concept to grasp as all of my Javascript experience had generally come with a DOM attached and usually aided by JQuery. It was difficult to see what I would do with this new construct and why I should use it.



A couple of weeks later, Hunter had just finished participating in the first Node Knockout. I was busy that weekend, and didn't know what I had missed at the time.



So back to the first event, Hunter's work in NKO had prompted him to build a simple multiplayer game demo to show off to the company again in another tech talk. The demo consisted of a small character in a 2D environment that you could move around.



The hook was that every user that connected to the app would get their own character and you could see all the characters moving and flying around in real time. The code he showed that it took to accomplish this was surprisingly simple and I was hooked.

102 lines of code

web server

web app

multiplayer game controller

```
1 contributor
file 105 lines (86 sloc) 2.095 kb
Edit Raw Blame History

1 var express = require('express'),
2   connect = require('connect'),
3   io = require('socket.io');
4
5
6 // Create and export Express app
7 var app = express.createServer();
8
9 app.set('development');
10
11 // Configuration
12 app.use(connect.bodyDecoder());
13 app.use(connect.methodOverride());
14 //app.use(connect.gzip());
15 app.use(connect.compiler({ src: __dirname + '/static', enable: ['sass'] }));
16 app.use(connect.staticProvider(__dirname + '/static'));
17
18 app.configure('development', function(){
19   app.set('reload views', 1000);
20   app.use(connect.errorHandler({ dumpExceptions: true, showStack: true }));
21 });
22
23 app.configure('production', function(){
24   app.use(connect.errorHandler());
25 });
26
27 // Routes
28
29 app.get('/', function(req, res) {
30   res.redirect('/index.html');
31 });
32
33
34 app.listen(3000);
35
36
37 // DEMO
38 // Socket.IO
39
40 function Game() {
41   this.players = [];
42 }
43 Game.prototype.summarize_for = function(client) {
44   var summary = {
45     type: 'game summary',
46     new_player_id: client.sessionId,
47     data: []
48   };
49   for(var i in this.players) {
50     summary.data.push(this.players[i].data);
51   }
52   client.send(json(summary));
53 };
54 Game.prototype.add_player = function(player) {
55   this.players.push(player);
56   player.client.broadcast(json({
57     type: 'player connected',
58     id: player.data.id
59   }));
60 }
61 Game.prototype.remove_player = function(player) {
62   player.client.broadcast(json({
63     type: 'player disconnected',
64     id: player.data.id
65   }));
66   this.players.splice(this.players.indexOf(player), 1);
67 }
68
69 function Player(id, client) {
70   var that = this;
71   this.client = client;
72   this.data = {
73     id: id,
74     x: 0,
75     y: 0,
76     dx: 0,
77     dy: 0,
78     sprite: 0
79   };
80   client.on('message', function(data) {
81     var obj = JSON.parse(data);
82     if (obj.type === 'player syno') {
83       that.data = obj.data;
84       client.broadcast(json({
85         type: 'player update',
86         data: that.data
87       }));
88     }
89   });
90   client.on('disconnect', function() {
91     game.remove_player(that);
92   });
93 }
94
95 var json = JSON.stringify,
96   io = io.listen(app),
97   game = new Game();
98
99 io.on('connection', function(client) {
100   game.summarize_for(client);
101   game.add_player(new Player(client.sessionId, client));
102 });
103
104
```

The code he showed that it took to accomplish this was surprisingly simple and I was hooked. I immediately pulled down the repo and starting playing.

Meet Canary.

The best way to keep your GitHub projects healthy.

519

users

1236

projects

183

commits

23220

files

All you need is a GitHub Account.

[Sign in & Get Started!](#)

I have joined Hunter, along with David and we've participated in the next 2 competitions since then. We placed 4th this year overall with our project Git Canary.



How did I get here?

What was the path that led to me working with Node as my preferred stack?

Ruby

.Net

Java

Python

PHP

HTML

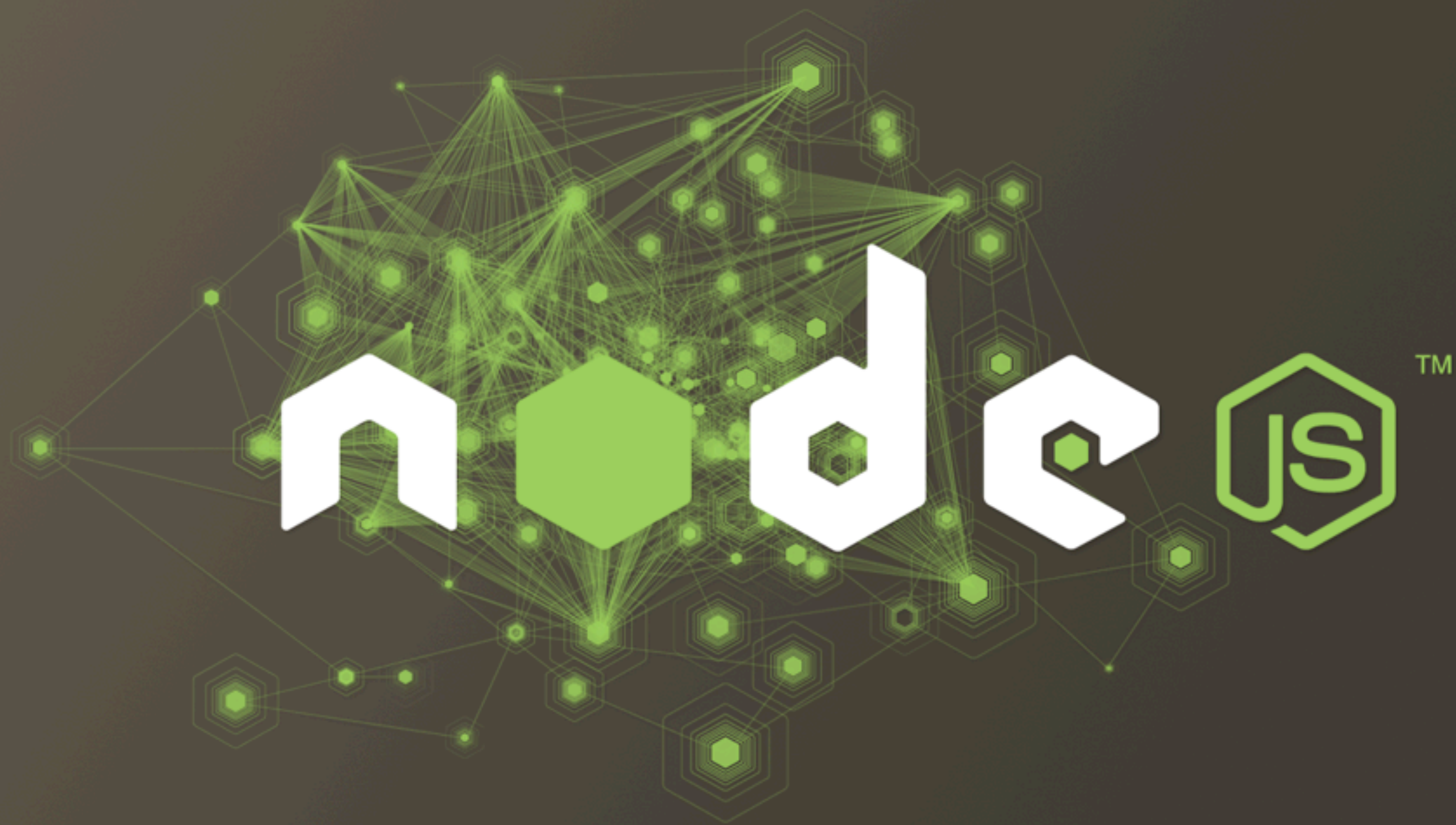
CSS

Flash

Actionscript

Javascript

Photoshop



Node is the culmination of everything else I have done.



Hello World

of course we have to start here

```
require('http').createServer(function handleRequest(req, res) {  
    res.writeHead(200, {'Content-Type': 'text/plain'});  
    res.end("Hello World");  
}).listen(1337, '127.0.0.1');  
  
console.log("web server listening on: 127.0.0.1:1337");
```

explain code at high level and why this is different than a typical PHP / Apache environment

introduce concept of require, console.log, callbacks, async workflow, events, etc



Hello World

the more useful version

```
var express = require('express');
var app = express();

// routing functions built in
app.get('/', function(req, res){

  // helper methods that set headers and send data for you
  res.send('Hello World');
});

app.listen(3000);
console.log('Listening on port 3000');
```

we'll talk about express later, but know it takes about the same amount of code as the native web server I showed, but comes with a lot of powerful features



Chatter

<https://github.com/snodgrass23/chatter>

chatroom demo built for this workshop

talk about how Chatter embodies many of the aspects that are so enjoyably about working with Node.js

```
Enter Your Username:      John

Chatroom: general chat

John : Hi everyone
Jim  : Where are the best parties at tonight?
Hunter : Count me in! I'm up for a party!
John : Alright, we should meet up somewhere.
John : █
```



javascript was used to write the clients seen here in a very short amount of time

```
var chatter = require('chatter');  
  
var chatter_server = new chatter.server({  
  port: process.env.PORT || 8000  
});
```

chatter code for creating a Node.js chat server

```
var chatter = require('chatter');
var chatter_client = new chatter.client("hostname.com");

// get last 10 messages in transcript
chatter_client.getRecentHistory();

// start listening for new messages
chatter_client.on('message', function(message) {
  console.log(message);
});
```

chatter api for a Node.js client

```
<script src="http://hostname.com/chatter/chatter.js"></script>
<script>
  chatter.connect('http://hostname.com', function(data) {
    console.log("handling new message: ", data);
  }, 500);

  chatter.getRecentHistory();

  chatter.send("Hello World!", "Client", function(response) {
    console.log("new message", response);
  });
</script>
```

chatter API for a front end JS client

<http://chatterjs.herokuapp.com/>

<http://skookum.github.com/WebChat/>

```
git clone git://github.com/snodgrass23/chatter.git  
cd chatter && npm install && make client
```

if you want to check out my test server or example apps
this is the URL for a test server
this is the URL for a demo mobile app David Becher made with the client lib
you can also clone the chatter repo and run an example server or client directly from
the terminal



Topics Overview

Javascript Concepts

Node Concepts

Node Basic Apps

Node Complex Web Apps and Deployments

overview of topics we'll attempt to cover



Javascript Concepts



Functions

```
function foo() { }

// realize that this actually creates an anon function
// that the var has a reference to
var bar = function() { };

// by naming the function, when exceptions are thrown
// inside the function debugging is made easier
// as the function name will be output with the exception
var baz = function baz() { };

// same concepts when declaring functions as properties of an object
var obj = {
  bar: function() { },
  baz: function baz() { }
}
```

function basics



Scope


```
var aNumber = 100;
tweak();

function tweak(){

    // This prints "undefined", because aNumber is also defined locally below.
    // the declaration is hoisted as such:
    // var aNumber = undefined;
    document.write(aNumber);

    if (false) {
        var aNumber = 123;
    }
}
```

Scope

explain hoisting

notice tweak is defined even though it is below its execution
aNumber is redefined locally inside

```
// common misuses
```

```
for (var i = 0; i < things.length; i++) {  
    var thing = things[i];  
};
```

```
if (thing) {  
    var newThing = thing;  
}  
else {  
    var newThing = {};  
}
```

incorrect assumptions of block scope



Closures

A closure is when a function closes around its current scope and that scope is passed along with the function object. This is accomplished by returning an inner function. That inner function retains all scope it had including any variables that were declared in its parent function.

```
function foo(x) {  
    var tmp = 3;  
  
    return function (y) {  
        alert(x + y + (++tmp));  
    };  
  
}  
  
var bar = foo(2); // bar is now a closure.  
bar(10);
```

bar has access to x local argument variable, tmp locally declared variable as well it's own y local argument variable

common use is for creating private methods and variables and only returning the public interface. this also helps keep variables in their own namespace to cut down on naming collisions, a common problem with Javascript's global variable model



Classes

```
function Person(options) {  
  options = options || {};  
  this.name      = options.name || "Jon Doe";  
  this.walk_speed = options.walk_speed || 5;  
  this.run_speed  = options.run_speed || 12;  
}
```

```
Person.prototype = {  
  speak: function speak() {  
    console.log(this.name + " speaks");  
  },  
  walk: function walk() {  
    console.log(this.name + " is walking at a speed of " + this.walk_speed);  
  },  
  run: function run() {  
    console.log(this.name + " is running at a speed of " + this.run_speed);  
  }  
};
```

```
var John = new Person({  
    name      : "John Doe",  
    walk_speed: 6,  
    run_speed  : 11  
});
```

```
John.walk();
```

```
// John Doe is walking at a speed of 6
```

```
John.run();
```

```
// John Doe is running at a speed of 11
```




Asynchronous Functions

```
// example DB class
function DBConnection() {
    this.run_query = function(query, callback) {
        setTimeout(function() {
            callback(null, query);
        }, 1000);
    };
}

function logQueryResults(err, result) {
    if (err) return console.log("ERROR:", err);
    console.log("Query Returned For: " + result);
}

// declare new connection variable with new instance of DBConnection
var connection = new DBConnection();
```

```
//request
```

```
console.log("handle http request");
```

```
connection.run_query("SELECT * FROM users", function(err, result) {  
    logQueryResults(err, result);  
});
```

```
//request
```

```
console.log("handle another http request for static asset");
```

```
//request
```

```
console.log("do some more cool stuff");
```



Async vs. Sync

In many languages, operations are all performed in a purely synchronous manner. The execution follows the code line by line waiting for each line to finish. In Javascript, you can also have functions that are executed asynchronously because of the way the event loop works.

```
$("#button").click(function() {  
    // do something when button is clicked  
});
```

You've all seen this when passing in a function to a click handler in JQuery. The code doesn't stop and wait for the click to happen before continuing, the JQuery click function sets up the proper events and returns immediately. It still has that callback function, though, that it can run whenever that event is triggered.

note the scope of the callback function will be of the element clicked

```
var messages = [  
  { username: "snodgrass23", body: "How's the dog doing?" },  
  { username: "dbecher", body: "Great, however my pillow is not so great after  
she ripped it apart" },  
  { username: "snodgrass23", body: "Have fun cleaning that up!" }  
];  
  
function getMessagesForUsername(username, callback) {  
  // some logic to iterate messages and return ones with requested username  
  callback(null, messages);  
}  
  
getMessagesForUsername("snodgrass23", function(err, messages1) {  
  getMessagesForUsername("dbecher", function(err, messages2) {  
    // do something with messages  
  });  
});
```

this seems like it should be async and non-blocking because it's using callbacks, right?

in reality, the event loop is blocked the entire time that all of this is running. This particular example would still run quickly, but in the real world, the message list would be much longer and there may be more computations that have to run with each result set which would also block.

```
setTimeout(function() {  
    // do something after 1ms second timeout  
}, 1);
```

```
process.nextTick(function() {  
    // do something on next cycle of the event loop  
});
```

<http://howtonode.org/understanding-process-next-tick>

One way to force an async function execution would be to add the code in a timeout or nextTick function. Those both say to wait an amount of time, which then allows the event loop to continue executing code.

```
require('fs').readFile('/var/logs/app', function (err, data) {  
  if (err) throw err;  
  console.log(data);  
});
```

In Node, async functions are particularly powerful as all IO runs through them, and IO is notoriously the largest blocker in most code, whether it's reading from a database or opening a local file. This is an extremely important concept in Nodes speed.



Callback Pattern

As mentioned before, this is simply the design pattern of passing in a function to another function (usually as the last argument) with the intent of that function being called when the called function is finished with it's task. Similar functionality can be handled with promises, and some people prefer to work with promises (they were a larger part of Node in early versions, but native Node methods have transitioned completely to callback and event patterns).

```
var UserModel    = require('./models/user'),
    CommentModel = require('./models/comment');

UserModel.findById(id, function(err, user) {

    CommentModel.find({username: user.username}, function(err, comments) {
        comments.forEach(function(comment) {
            console.log(comment);
        });
    });
});
```

note consistent arguments returned from each method
note lack of error handling



ECMAScript Latest

Node uses the latest build of V8 which has all of the newest features of ECMAScript 5.1 and even some limited features from ES6 when using the `--harmony` flag.

```
node -e 'console.log(process.versions)'
```

```
{ http_parser: '1.0',  
  node: '0.8.18',  
  v8: '3.11.10.25',  
  ares: '1.7.5-DEV',  
  uv: '0.8',  
  zlib: '1.2.3',  
  openssl: '1.0.0f' }
```

<http://v8.googlecode.com/svn/trunk/ChangeLog>

this command will allow you see the embedded versions of things in your current node build. this particular output was what I got when I was on Node.js v0.8.18

I could then look through v8's change log to see exactly what features I have

```
/**
 * Getting the keys of an object
 * without iterating through and using hasOwnProperty
 */
var book = {
  name      : "The Hitchhiker's Guide to the Galaxy",
  author    : "Douglas Adams",
  num_pages: 224
};

Object.keys(book);
//[ 'name', 'author', 'num_pages' ]
```

```
/**  
 * accurately checking for an array type  
 */  
var array = [];  
  
var type = typeof array1;  
// object  
  
Array.isArray(array1);  
// true
```

```
/**
 * native array iteration
 */

["jim", "david", 23].forEach( function(value) {
    console.log(value);
});
// jim
// david
// 23
```

```
/**
 * native array filter
 */

["jim", "david", 23].filter( function(value) {
    return typeof value == "string";
});
// [ 'jim', 'david' ]
```



```
/**  
 * Native string trim  
 */  
  
"  hello world ".trim();  
// 'hello world'
```

```
/**
 * function binding
 */
function a() {
  return this.hello == 'world';
}

a();
// false

var b = { hello: 'world' };

var c = a.bind(b); // sets scope of 'this' to b

c();
// true
```

```
// Accessor methods (__defineGetter__, __defineSetter__)
Date.prototype.__defineGetter__('ago', function() {
    var diff = new Date() - this;
    var conversions = [
        ["years", 31518720000],
        ["months", 2626560000 /* assumes there are 30.4 days in a month */],
        ["days", 86400000],
        ["hours", 3600000],
        ["minutes", 60000],
        ["seconds", 1000]
    ];

    for (var i = 0; i < conversions.length; i++) {
        var result = Math.floor(diff / conversions[i][1]);
        if (result >= 2) return result + " " + conversions[i][0] + " ago";
    }
    return "just now";
});

var my_bd = new Date('3/24/1978');

my_bd.ago; // '34 years ago'
```



Native JSON

```
JSON.stringify(obj);
```

```
JSON.parse(string);
```

```
var users = require('./users.json');
```

Node has JSON handling methods built in: stringify, parse. It can also load in a JSON file using the require() syntax. The file will be loaded as a Javascript object parsed from the JSON string.



Questions?



Node.js Concepts



Why Node.js?

Lets first talk a little about why Node.js is a desirable environment to develop in. What exactly does it give you that's different than many other web languages?

Asynchronous IO

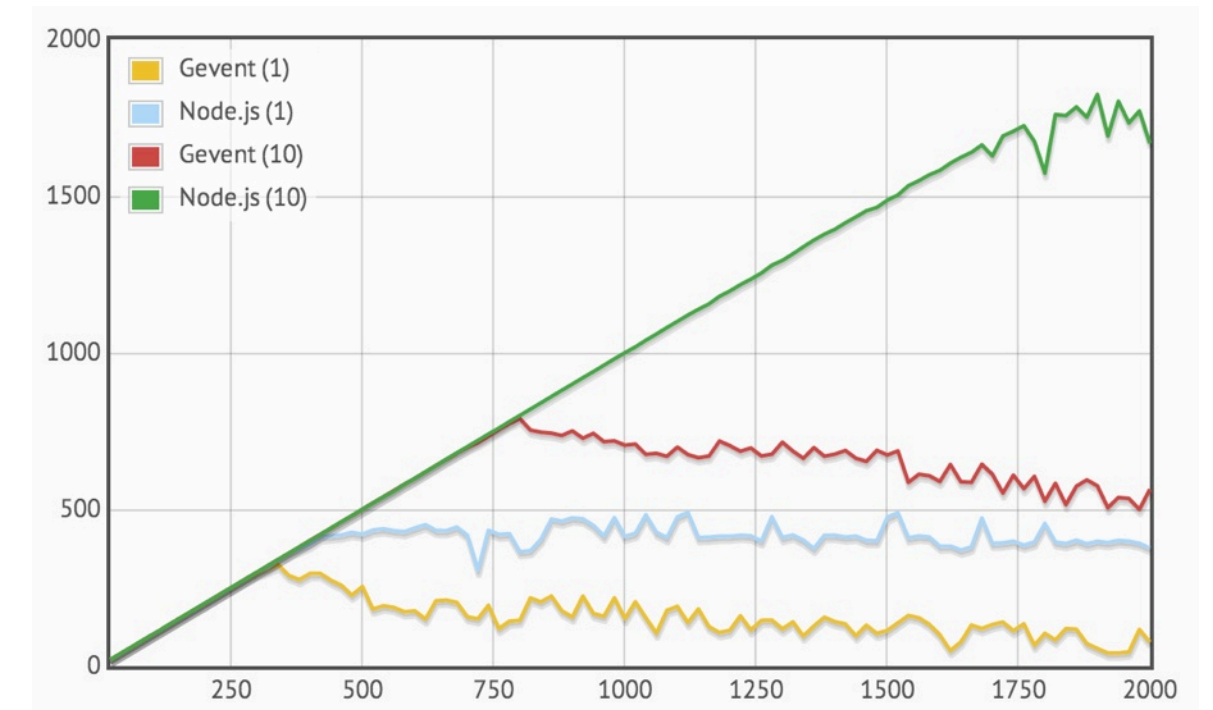
database access
file access
network access



IO operations are some of the slowest things an app can do. The app sends a request and waits around to get a response back. With Node.js, these operations are all done asynchronously, which means the app can keep working while those operations are being performed.

Speed

high requests per second
fast response time
fast startup time



Preference

it's just Javascript

use familiar libraries

easy to move between levels of app

easy integration with html/css preprocessors





Single process and memory state

```
var books = [  
  { title: "Smashing Node.js", author: "Guillermo Rauch" },  
  { title: "Javascript: The Good Parts", author: "Douglas Crockford" },  
  { title: "Eloquent Javascript", author: "Marijn Haverbeke" }  
];  
  
function handleRequest(req, res) {  
  var output = "All Books: \n", book;  
  while(books.length) {  
    book = books.pop();  
    output += book.title + " by " + book.author + "\n";  
  }  
  res.writeHead(200, {'Content-Type': 'text/plain'});  
  res.end(output);  
}  
  
var http = require('http');  
http.createServer(handleRequest).listen(1337, '127.0.0.1');
```

this is an example of a basic web app that might not behave as you would expect. the first time a user visits the page, it will be fine. All other times afterward the list of books will be empty.

```
var start = Date.now();

setTimeout(function() {
  console.log(Date.now() - start);
}, 1000);

setTimeout(function() {
  lotsOfProcessing();
  console.log(Date.now() - start);
}, 2000);

function lotsOfProcessing() {
  var highest_number = 0;
  for (var i = 0; i < 10000000000; i++) {
    if (i > highest_number) {
      highest_number = i;
    }
  }
}
```

notice the first timeout will trigger the console log at @1002ms when the event loop gets to it
the second timeout will do the same
then, if we add some complex processing to the second one, the first log will still trigger at about the same time, but the second one will be probably a second or so later because it has to wait for that function to finish.



Event Loop

we'll go over some more concepts that we've already touched on, but these are the most important things to grasp for effective Node development


```
console.log("first");

UserModel.findById(id, function(err, user) {
  console.log("third");
  CommentModel.find({username: user.username}, function(err, comments) {
    comments.forEach(function(comment) {
      console.log(comment);
    });
  });
});

console.log("second");
```

Everything in Node runs on the event loop. When a function is called it is added to the call stack and handled as the event loop gets to it. In the basic sense this is similar to a synchronous language if we also handled IO calls the same way they do. However, functions have the option of adding something to the stack for later processing, either on the next iteration or later. The event loop is then able to continue handling items in the call stack instead of waiting for that task to finish. When the task finishes, its callback is added to the stack and handled appropriately.


```
function lotsOfProcessing() {  
  var highest_number = 0;  
  for (var i = 0; i < 10000000000; i++) {  
    if (i > highest_number) {  
      highest_number = i;  
    }  
  }  
}
```

don't do things synchronously that will hold up the event loop and block



Handling Errors

Because of the way the call stack and event loop work, it's very important to handle errors appropriately at every step of the way. If an error is uncaught, it causes an uncaught exception. Because the system has no way of knowing if the state of the app is valid at this point, the uncaught exception error will be thrown and the application will be halted.

```
function process(callback) {  
  setTimeout(function(){  
    callback(x+y);  
  }, 0);  
}  
  
process(function(){});
```

timers.js:103

```
if (!process.listeners('uncaughtException').length) throw e;  
                                                    ^
```

ReferenceError: x is not defined

```
at Object._onTimeout (/Users/jimnodgrass/Sites/node/sxsw node workshop/node_concepts/stack_trace.js:3:14)  
at Timer.list.ontimeout (timers.js:101:19)
```

The asynchronous nature of Node can make these errors tricky to deal with. For example, in a synchronous application, the call stack will be a traceable series of calls that lead to the error. In Node, an asynchronously called function will not be next to its contextual elements when it is called and the stack trace will be invalid and not even show in the error message.

```
function causeError() {  
  console.log(x+y);  
}  
  
function process(callback) {  
  setTimeout(causeError, 0);  
}  
  
try {  
  process(function(){});  
}  
catch(e) {  
  console.log("caught error: ", e);  
}
```

timers.js:103

```
if (!process.listeners('uncaughtException').length) throw e;  
                                                         ^
```

ReferenceError: x is not defined

```
at Object.causeError (/Users/jimsnodgrass/Sites/node/sxsw node workshop/node_concepts/stack_trace.js:2:15)  
at Timer.list.ontimeout (timers.js:101:19)
```

Another thing to realize is that a try-catch statement doesn't work on async functions. Once the initial function call finishes and allows the event loop to continue, it will move through the try-catch block and continue on. If that function throws an exception on a later iteration of the event loop, that catch statement is not around anymore to catch it. It's because of this that Node relies heavily on asynchronous methods of getting these events.



Callback standards

```
function add(x, y, callback) {  
  setTimeout(function() {  
    if(!x || !y) return callback(new Error("missing arguments"));  
    callback(null, x+y);  
  }, 0);  
}
```

```
add(undefined, undefined, function(err, result){  
  if (err) console.log(err);  
  else console.log(result);  
});
```

```
[Error: missing arguments]
```

The standard way for native Node methods, and many libraries, to handle errors is with a standard callback function argument signature. There are two arguments, the first always being the error and the second the result of the function. Therefore, you want to check this error argument anytime you call an async function so that an uncaught exception isn't thrown and you can accurately handle the error without bringing down the entire app.

It's also important for the functions to accurately determine when to pass an argument back through the callback function

```
User.findById(482726, function(err, user1) {  
  if (err) {  
    callback(err);  
  }  
  else {  
    User.findById(974253, function(err, user2) {  
      if (err) {  
        callback(err);  
      }  
      else {  
        User.findById(6345928, function(err, user3) {  
          if (err) {  
            callback(err);  
          }  
          else {  
            User.findById(813468, function(err, user4) {  
              callback(err, [user1, user2, user3, user4]);  
            });  
          }  
        });  
      }  
    });  
  }  
});  
});
```

Another issue that arises with many new Node devs is they end up with huge "pyramids" of code. They end up with a function that calls an async function, which then checks for errors with an if/else statement, which then calls another async function and so forth. You can very quickly end up with what I like to call the "pyramid of doom."

```
User.findById(482726, function(err, user1) {  
  if (err) return callback(err);  
  User.findById(974253, function(err, user2) {  
    if (err) return callback(err);  
    User.findById(6345928, function(err, user3) {  
      if (err) return callback(err);  
      User.findById(813468, function(err, user4) {  
        callback(err, [user1, user2, user3, user4]);  
      });  
    });  
  });  
});
```

There are a couple ways we combat this temptation. One way is to use an early return statement on errors, which removes the requirement of an "else" block, therefore removing one level of the pyramid for each call.


```
async.parallel({
  user1: function(callback){
    User.findById(482726, done);
  },
  user2: function(callback){
    User.findById(974253, done);
  },
  user3: function(callback){
    User.findById(6345928, done);
  },
  user4: function(callback){
    User.findById(813468, done);
  }
}, function(err, results) {
  // err and results of all tasks
  // results = {user1: {}, user2: {}, user3: {}, user4: {}},
});
```

Another way is to use a control flow pattern or library that allows you to list the functions in order with a final callback that handles any errors throughout the other calls and receives a final resulting object.

```
function getUsers(users, callback) {  
  var results = [];  
  user.forEach(function(user) {  
    User.findById(user, function(err, result) {  
      if (err) return callback(err);  
      results.push(result)  
      if (results.length == user.length) callback(null, results);  
    });  
  });  
}  
  
getUsers([482726, 974253, 6345928, 813468], function(err, users) {  
  // got users or error  
})
```

The best way to handle this tangle of functions is just to be very cognizant of this tendency when planning your code. I've found that this comes almost naturally with experience if the developer is aware of it.



Events and Promises

```
fs.stat("foo", function (error, value) {  
  if (error) {  
    // error  
  }  
  else {  
    // ok  
  }  
});
```

```
var promise = fs.stat("foo");  
promise.addListener("success", function (value) {  
  // ok  
})  
promise.addListener("error", function (error) {  
  // error  
});
```

<http://howtonode.org/promises>

Another way to handle errors is with events and promises. In early versions, Node used promises extensively with many of its native methods. They have since moved to the callback standard, but promises remain a popular alternative to many things and there are libraries out there that will help you use any function, even native methods, with promises. Personally, I don't use them much as I like the consistency of just doing things the same way Node does them.

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

function Server(options) {
  this.options = options || {};
}

require('util').inherits(Server, EventEmitter);

var chat_server = new Server();

chat_server.on('message', function(message) {
  // handle message
});

chat_server.on('error', function(error) {
  // handle error
});
```

There will be times when it is much easier and cleaner to use an event driven system of catching and throwing errors. This is common when there may be many listeners that need to know about an error's occurrence so that it can be handled cleanly.



Streams

```
client.get('/test/Readme.md').on('response', function(res){  
  console.log(res.statusCode);  
  console.log(res.headers);  
  res.setEncoding('utf8');  
  res.on('data', function(chunk){  
    console.log(chunk);  
  });  
}).end();
```

<https://github.com/LearnBoost/knox>

This is an example of a stream using Knox to pull down a file from Amazon S3.

notice the event listener on “response” and “data”



Node Modules

Node modules are key to Node's usefulness as many different types of apps and servers. All of Node's native methods are handled as modules, such as the `net` and `fs` modules which give the app access to many methods contained within them. For those familiar with Rails, Node's modules are similar to gems in Ruby. Modules are really a requirement for a server application written in Javascript which would have an extremely cluttered global namespace if libraries were handled like client side Javascript where each library offered a global accessor variable (think JQuery or \$).



```
$ npm install chatter  
$ npm install -g express
```

NPM is the package manager used to handle the many 3rd party modules your app may need. It is included in the Node.js installers
You can manually install any package by simply typing `npm install` and the package name. The module will be installed in a self named directory inside a `node_modules` directory. You can then use the module in your app using the `require` syntax and the name of the module.

```
{
  "name": "chatter",
  "description": "module for building chat server and client",
  "version": "0.0.1",
  "author": "Jim Snodgrass <jim@skookum.com>",
  "engines": {
    "node": ">=0.8.0",
    "npm": ">=1.1"
  },
  "dependencies": {
    "express": "3.x",
    "superagent": "latest",
    "underscore": "latest"
  },
  "repository": "https://github.com/snodgrass23/chatter",
  "main": "index"
}
```

You can also create package.json file in the root of your app and simply run 'npm install' and npm will download and, as needed, compile all required modules. Each of those modules will also have a package.json file that will specify it's own required modules as well as meta data like name and version. NPM will also install all of the dependent modules for those as well until every module has all of its dependents available.

```
var chatter = require('chatter');  
var chatter_client = new chatter.client("hostname.com");  
  
// get last 10 messages in transcript  
chatter_client.getRecentHistory();  
  
// start listening for new messages  
chatter_client.on('message', function(message) {  
    console.log(message);  
});
```

To use any of the modules installed with npm or any of Node's native modules, simply use the require function using the name of the module as the only argument. You'll generally assign the return of that require statement to a variable which you can then use to call any of the methods available from the module. Here is the original example for the chatter module and you can see how we are using it here.

```
// this file saved as classroom.js

var students = [ "Jim", "David", "Hunter", "Dustan", "Jason", "Bryan" ];

exports.students = students;
exports.getStudent = function getStudent(id) {
  return students[student_index] || null;
}


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

classroom.students;
classroom.getStudent(2) // Hunter
```

You can also build your own modules, whether large or small, in your app and use require to access their methods as well. The only difference is you need to include the relative path to the main module file when requiring the module. Every module that is accessed via require has a module and exports property globally available to them. The exports object will be what is return to the code that is requiring them.

```
function Classroom() {  
  this.students = [ "Jim", "David", "Hunter", "Dustan", "Jason", "Bryan" ];  
}  
  
Classroom.prototype = {  
  getStudents: function getStudents(callback) {  
    return callback(null, this.students);  
  },  
  getStudentById: function getStudentById(id, callback) {  
    if (!this.students[id]) return callback(new Error("Student not found"));  
    return callback(null, this.students[id]);  
  }  
};  
  
module.exports = new Classroom();
```

The exports property is actually just an object on the module property, so if you'd rather return a function you can overwrite the exports object with it. This is a common way of exporting constructors on a class.

```
{
  "name": "express",
  "description": "Sinatra inspired web development framework",
  "version": "3.1.0",
  "author": {
    "name": "TJ Holowaychuk",
    "email": "tj@vision-media.ca"
  },
  "dependencies": { },
  "main": "index",
  "bin": {
    "express": "./bin/express"
  }
}
```

Some modules can even include a binary that can be accessed through the CLI. When installing these modules, use the global flag to make them accessible throughout your environment. Express, a popular web framework, is one that also includes simple app generators that you can use if you install it globally. We'll go over using some of these programs a little later on.



Should I use this module?

express

Sinatra inspired web development framework

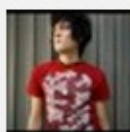
```
$ npm install express
```

12 992 downloads in the last day

108 680 downloads in the last week

366 146 downloads in the last month

Maintainers




tjholowaychuk

Version

3.1.0 last updated a month ago

When looking at modules to use, there are many things you need to consider. One of the first things I look at is the age of the library and more specifically when was it last updated. Node has been going through versions rapidly and a module that hasn't been updated for a year may be many versions old and may not even be compatible with current versions of Node and NPM.

 visionmedia / **express**

Watch

Star

9,046

Fork

1,237

Code

Network

Pull Requests31

Issues102

Wiki

Graphs

Sinatra inspired web development framework for node.js -- insanely fast, flexible, and simple — [Read more](#)
<http://expressjs.com>

Clone in Mac

ZIP

HTTP

SSH

Git Read-Only

git://github.com/visionmedia/express.git

Read-Only access


branch: master


Files

Commits


Branches8


Tags102

express / 

 **1000+ commits**

Merge pull request **#1533** from shesek/old-viewcallbacks ...

 **visionmedia** authored 4 days ago

latest commit 9df93d6dec 

I also like to look at the Github page for the module and look at things like the reported issues. This tells me how active and responsive the contributors are to real user's issues as well as changes to Node versions and standards. I'll also consider the type of library it is when looking at it's activity. If its a library dealing with Dates or Numbers it may be stable already and not need a lot of upkeep.



Questions?



Node.js Basic Apps



CLI Apps

Node may not be the first thing many people think of when writing a basic CLI script or app, but it actually works quite well. Node is able to access all of the arguments passed in through CLI using the `process.argv` array. The first two items are always node itself and the path of the script being executed. You can easily slice off these elements to get just an array of the rest of the arguments passed in.

```
#!/usr/bin/env node

var program = require('commander');

program
  .version('0.0.1')
  .option('-p, --peppers', 'Add peppers')
  .option('-P, --pineapple', 'Add pineapple')
  .option('-b, --bbq', 'Add bbq sauce')
  .option('-c, --cheese [type]', 'Add the specified type of cheese [marble]',
'marble')
  .parse(process.argv);

console.log('you ordered a pizza with:');
if (program.peppers) console.log('  - peppers');
if (program.pineapple) console.log('  - pineappe');
if (program.bbq) console.log('  - bbq');
console.log('  - %s cheese', program.cheese);
```

<https://github.com/visionmedia/commander.js/>

In a production app with the need for arguments and flags, I would recommend using the [Commander.js](<https://github.com/visionmedia/commander.js/>) node module.

```
$ ./order_pizza --help
```

```
Usage: order_pizza [options]
```

```
Options:
```

```
-h, --help          output usage information
-V, --version        output the version number
-p, --peppers        Add peppers
-P, --pineapple      Add pineapple
-b, --bbq            Add bbq sauce
-c, --cheese [type]  Add the specified type of cheese [marble]
```

```
$ ./order_pizza
```

```
you ordered a pizza with:
```

```
- marble cheese
```

```
$ ./order_pizza -b -c mozzarella
```

```
you ordered a pizza with:
```

```
- bbq
- mozzarella cheese
```



Express apps

```
npm install -g express
express --sessions --css stylus myapp
cd myapp && npm install && node app
```

Express.js is one of the main frameworks in use today for building web apps in Node and was built on top of connect. Express handles the http requests that come into node as well as all of the framework that needs to accompany a web app like routing, html view templating, and serving static files.



Questions?



Node.js Complex Web Apps and Deployments



Common DB Choices



MongoDB

Mongo is my preferred data store. It just fits really well into the Node.js framework and I really like the ability to define my models and schemas as needed in my code as opposed to building them into the database as a separate action.

```
var MongoClient = require('mongodb').MongoClient;

// Connect to the db
MongoClient.connect("mongodb://localhost:27017/exampleDb", function(err, db) {
  if(err) { return console.dir(err); }

  var collection = db.collection('test');
  var doc1 = {'hello':'doc1'};
  var doc2 = {'hello':'doc2'};
  var lotsOfDocs = [{'hello':'doc3'}, {'hello':'doc4'}];

  collection.insert(doc1);

  collection.insert(doc2, {w:1}, function(err, result) {});

  collection.insert(lotsOfDocs, {w:1}, function(err, result) {});

});
```

<http://mongodb.github.com/node-mongodb-native/>

You can use the mongodb driver directly, creating collections and inserting items into them.

```
var mongoose = require('mongoose');
mongoose.connect('localhost', 'test');

var schema = mongoose.Schema({ name: 'string' });
var Cat = mongoose.model('Cat', schema);

var kitty = new Cat({ name: 'Zildjian' });
kitty.save(function (err) {
  if (err) // ...
    console.log('meow');
});
```

<http://mongoosejs.com/>

My preference is to use Mongoose.js. It gives you a nice interface for modeling objects and handling them in a very natural way.

Person

```
.find({ occupation: /host/ })  
.where('name.last').equals('Ghost')  
.where('age').gt(17).lt(66)  
.where('likes').in(['vaporizing', 'talking'])  
.limit(10)  
.sort('-occupation')  
.select('name occupation')  
.exec(callback);
```

example of some of the query methods Mongoose gives you.

```
var toySchema = new Schema({
  color: String,
  name: String
});

var Toy = mongoose.model('Toy', toySchema);

Toy.schema.path('color').validate(function (value) {
  return /blue|green|white|red|orange|periwinkle/i.test(value);
}, 'Invalid color');

var toy = new Toy({ color: 'purple' });

toy.save(function (err) {
  // err.errors.color is a ValidatorError object

  console.log(err.errors.color.message)
  // prints 'Validator "Invalid color" failed for path color'
  console.log(err.name) // prints "ValidationError"
  console.log(err.message) // prints "Validation failed"
});
```

mongoose model with validation



Redis

Redis is a really fast key-value store that is great for things like managing sessions or handling custom caching needs.

```
var redis = require("redis"),
    client = redis.createClient();

client.on("error", function (err) {
  console.log("Error " + err);
});

client.set("string key", "string val", redis.print);

client.get("key", function(err, reply) {
  // reply is null when the key is missing
  console.log(reply);
});
```

You can use the redis node module to manually connect to a redis instance. You can then create a client that will allow you to set and get items from the store. I've used Redis directly like this many times to save a cache for processed data to improve performance.

```
var express = require('express'),  
    app = express(),  
    RedisStore = require('connect-redis')(express);  
  
app.use(express.session({ secret: "mysessionsecret", store: new RedisStore }));
```

The most common way I use redis is as a session store. This will be a persistent way to store your session data, so that multiple apps can share the sessions and so that when the app restarts it doesn't lose all the sessions like it would if they were just stored in memory, which is the default.



MySQL

If you'd like to you can use MySQL as your main data store,
or maybe you have a legacy DB that you need to connect to

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host        : 'localhost',
  user        : 'me',
  password    : 'secret',
});

connection.connect();

connection.query('SELECT 1 + 1 AS solution', function(err, rows, fields) {
  if (err) throw err;

  console.log('The solution is: ', rows[0].solution);
});

connection.end();
```

here is an example of using the mysql module to connect to a db and run a query

```
var sequelize = new Sequelize('database', 'username', 'password', {
  host: "my.server.tld",
  port: 12345
})

var Project = sequelize.define('Project', {
  title: Sequelize.STRING,
  description: Sequelize.TEXT
})

// search for known ids
Project.find(123).success(function(project) {
  // returns rown with id 123. if such an entry is not defined you will get null
})

// search for attributes
Project.find({ where: {title: 'aProject'} }).success(function(project) {
  // project will be the first matchhing entry 'aProject' || null
})
```

<http://sequelizejs.com/>

Sequelize is another module that gives you more ORM capabilities allowing you define models and queries in a more natural Javascript way.



Scaling

I need more processes!!

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

module.exports = function balance(init, max) {
  return cluster.isMaster? initMaster() : init();
};

function initMaster() {
  cluster.on('exit', function(worker) {
    cluster.fork();
  });

  cluster.on('death', function(worker) {
    cluster.fork();
  });

  var workerCount = process.argv[3] || os.cpus().length;
  var i = workerCount;
  while(i--) {
    cluster.fork();
  }
}
```

the first hurdle is how do you start multiple processes. There have been many ways in the past to do this, but starting in version 0.6?? Node has a native API for this called cluster.


```
var express = require('express'),
    app = express(),
    RedisStore = require('connect-redis')(express);

app.use(express.session({ secret: "mysessionsecret", store: new RedisStore }));
```

the next problem you run into when attempting to scale past one process is that by default many things like sessions will be depend on the shared memory of the app. unfortunately, the other instances of your app will not have access to these items. This is where redis comes in. So, like I mentioned before about using Redis as your session store, all of the app instances will be able to share that data.



Deployments



Heroku

Heroku is awesome for putting up apps real quick with minimal overhead. The chatter app, for example, is sitting on a free Heroku instance. Deployment is as easy as pushing the repo up.

```
# Procfile
```

```
web: node app.js simple
```

```
var chatter = require('../index');
```

```
var chatter_server = new chatter.server({  
  port: process.env.PORT || 8000  
});
```

When deploying to heroku, there are just a few things you need to be aware of. You need to first have a Procfile. You place this in the root of your app and Heroku will use the information in this to determine how to launch your app. You should also be aware that all config items will be stored as environment variables.



Linode

Linode is my goto right now for a full production app. It allows me to manage my server in whichever way I want. I can have my own DB without any connection or size limits and I can run whatever language I want. I also can run 4 processes optimally to match the 4 cores on my server. The cost for this as compared to something like Heroku with 4 dynos and separate accounts for any databases like Mongo or Redis is considerably less expensive. The downside is that you have to manage provisioning and deployments manually.



Questions?

https://github.com/Skookum/sxsw13_nodejs_workshop

sxsw.tv/cj2

jim@skookum.com
[@snodgrass23](#)