Busy Developer's Guide to NodeJS

Ted Neward

Neward & Associates

http://www.tedneward.com | ted@tedneward.com

Credentials

Who is this guy?

- CTO, iTrellis (http://www.itrellis.com)
  - **ask me how we can help your project, your team or your firm**
- Microsoft MVP (F#, C#, Architect); JSR 175, 277 EG
- Author
  - **Professional F# 2.0 (w/Erickson, et al; Wrox, 2010)**
  - **Effective Enterprise Java (Addison-Wesley, 2004)**
  - **SSCLI Essentials (w/Stutz, et al; OReilly, 2003)**
  - **Server-Based Java Programming (Manning, 2000)**
- Blog: http://blogs.tedneward.com
- Writing: http://www.newardassociates.com/writing.html
- Twitter: @tedneward
- For more, see http://www.newardassociates.com/about.html

Objectives

- See how to get started with NodeJS
- See some NodeJS example code
- Explore some NodeJS modules
- Discuss its pros/cons over other tools

NodeJS Basics

Because we have to start somewhere

Starters

NodeJS is JavaScript... on the server

- Yes, that's really all it is
- Actually, it's Javascript outside of the browser
  - **on the command-line**
  - **in the NoSQL database**
  - **in your web server**
  - **on the server itself (outside of the web server)**
  - **anywhere else you can embed the V8 engine**

Starters

So... why does everyone care?

- Because JavaScript is hot
- (Seriously, that's pretty much it)

Starters

So... why does everyone care?

- Because JavaScript is hot
- JavaScript-on-the-client, JavaScript-on-the-server
- Scripting language vs "system language" (Java, C#, etc)
- Lighter-weight stack (Node vs JavaEE, Node vs .NET)

Truthfully, half the magic is in the packages (npm)

Starters

So... what does Node really look like?

- Basically, just JavaScript without all the DOM stuff
- No "main()", just start executing from the top

Starters

Hello, Node

```
console.log("Hello, node!")
```

NodeJS Installation

Because we have to start somewhere

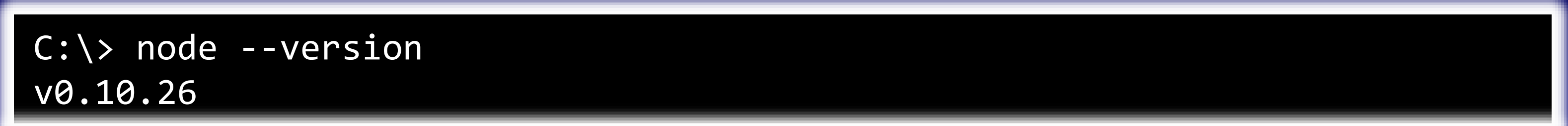
Getting Started

Installing Node

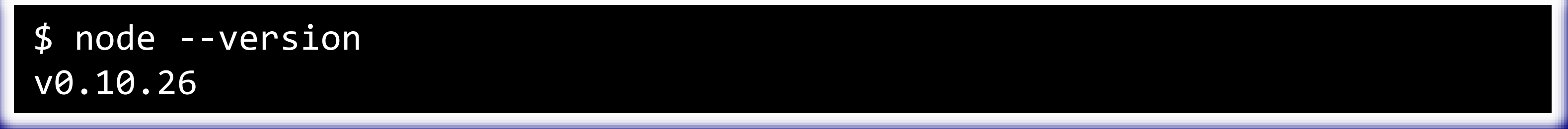
- http://nodejs.org
    - **Windows: MSI install**
    - **MacOS X: DMG install**
- Platform-specific
    - **Windows: WebPI**
    - **MacOS X: brew install node**

Getting Started

Verifying it's there



```
C:\> node --version
v0.10.26
```



```
$ node --version
v0.10.26
```

Getting Started

Hello, node

```
console.log("Hello, node!")
```

```
$ node helloNode.js
Hello, node!
```

Starters

At heart, Node the Google V8 engine

- tons of command-line flags available
- most of them are irrelevant or trivial or esoteric
- get a list with "--v8-options" if you wish

Starters

Node runs as a REPL

- help: Brings up help
- break: Abort current command mode (get back to prompt)
- clear: Clear REPL content
- exit: Out we go

{ECMA|Java}Script Review

Because NodeJS IS JavaScript

Overview

ECMAScript has ...

- ... an imperative C-family-of-languages syntax
- ... a classless object system
- ... functional objects
- ... loosely-typed type system
- ... a metaobject protocol
- ... a garbage collected object heap
- ... and a few bad design decisions/legacy

Basics

Starting points

- Whitespace: space, tabs, CRLF, etc

  mostly irrelevant
  line terminator (";") mostly optional
- Comments: // (end-of-line) and /* */ (multi-line)
- Identifiers/Names: [A-Za-z][A-Za-z0-9...]
- Numbers: always a 64-bit floating-point type, NaN
- Strings: 16-bit Unicode

Variables

Variables

- signified with "var" keyword, followed by legal name
- any variable used without "var" declaration is global

  this is generally considered bad
  be particularly careful in "for" loops
- variables are typeless

  but the things they point to are typed
  just not very strongly; coercion is always possible

Flow control

Flow control primitives familiar to C-family langs

- if/else, switch/case, try/catch, while, do/while, for
  - **"for (a in b)" is an iterating for**
- test expressions are evaluated for "truthiness"
  - **'falsy' values: false, null, undefined, '', 0, NaN**
  - **'truthy' values: anything else**
- labels are similar to C-family syntax
  - **name: at the start of any line in a block**
  - **break is a labeled break**
  - **break; (exits current scope) or break label; (break to label)**
- return always yields a value (undefined if nothing else)
- throw starts popping execution records looking for catch

Operators

Operators

- operator set similar to that from C-family langs

  but there are some subtle and dangerous differences!
- + - * / % : mathematical operators
- <= >= != < > : comparison operators
- === !== : equality/inequality operators

  ES also supports == and !=, but they attempt conversion
- && || ! : logical operators
- typeof : returns type of object

  object, function, undefined, Number, String, ...

Operators

What's truthy? What's falsy?

```
0 == ''
'' == '0'
false == '0'
false == null
null == undefined
false == undefined
```

Operators

What's truthy? What's falsy?

```
0 == ''                    (true)
'' == '0'                  (false)
false == '0'               (true)
false == null              (false)
null == undefined          (true)
false == undefined         (false)
```

Operators

Operators

- . [] () : "refinement" and "invocation" operators
- any use of "." or "[]" is an attempt to refine (find a property)
- any use of "()" is an attempt to invoke

this is extremely powerful; we'll see this again later

Functions

Functions are first-class citizens in ES

- functions are objects, too
- composition: 4 parts

  "function"
  name (optional)
  parameter set (0 - n named arguments)
  statement block
- function can appear anywhere an expression is expected

  top-level, as object members, nested, and so on
- two implicit arguments to every function invocation

  'this': reference whose contents vary with invocation pattern
  'arguments': array of arguments passed in
- unlike other languages, functions don't enforce arity

  missing arguments are undefined, extras are in 'arguments'

Functions

Functions

```
function addIt(first, second) {
  return first + second
}
println(addIt(1, 2))

var addItAgain = function(first, second) {
  return first + second
}
println(addItAgain(1,2))

println(function(first, second) {
  return first + second
}(1, 2))

var add = function() {
  var result = 0;
  for (var i = 0; i<arguments.length; i++)
    result += arguments[i]
  return result
}
println(add(1, 2, 3, 4, 5))
```

Functions

Function invocation patterns

- Function Invocation: function is not an object member

  "this" is bound to the global object
- Method Invocation: function is an object member

  "this" is bound to object on which function is being invoked
- Apply Invocation: function is invoked explicitly via apply()

  "this" is bound to first argument in parameters list
- Constructor Invocation: function is used as a constructor

  new object created with hidden link to function's prototype
  "this" is bound to newly-created object
  (this style is discouraged; embrace prototypical construction)

Functions

Function scope

- ES is not block-scoped, as C-family languages are

  suggestion: declare vars before use at top of function
  suggestion: prefer functions, not blocks

- nested functions get access to outer function scope

  known as "closure": variables referenced in nested function survive as long as inner function does

Functions

Function scope

```
function badScope() {
  for (var i = 0; i < 10; i++) {
    for (var j = 0; j < 10; j++) {
      var i = i * j
      println(i)
    }
  }
}
//badScope() // never terminates!

function goodScope() {
  for (var i = 0; i < 10; i++) {
    (function () {
      for (var j = 0; j < 10; j++) {
        (function(i, j) {
          var i = i * j
          println(i)
        })(i, j);
      }
    })();
  }
}
goodScope();
```

Objects

Objects are essentially a bag of name-value pairs

- values can be either data or function values
- classless system: no concept of "class", just "objects"
- "open" objects: members can be added/removed
- members accessed through refinement operators (. [])
- use [] to access illegal identifier names (as keys)

Objects

Objects

```
var speaker = {
  'firstName' : 'Ted',
  'lastName' : 'Neward',
  sayHello : function() {
    println("Hello!")
  },
  sayHowdy : function() {
    println("Howdy!")
  }
}
println(speaker.firstName)
println(speaker["lastName"])
speaker.sayHowdy()
speaker["sayHello"]()

for (var m in speaker) {
  println(m + "=" + speaker[m])
}
```

Objects

Object prototypes

- objects always have a "prototype" object
- prototype is always in scope when resolving names
- this creates a "prototype chain" of names
- we can control the prototype used at construction ...

  ... but the syntax for doing so in ECMAScript is... complicated.
- instead, monkey-patch Object and add a create() method

Objects

Objects and prototypes

```
var empty = { }
for (var m in empty) {
  println(m + "=" + empty[m])
}
println(empty.toString())
```

Objects

Monkey-patched Object.create:

- this version explicitly creates empty object, then links it to the prototype object passed in
- doesn't change Object.prototype, however, localizing the change (which is also important)

Objects

Monkey-patching

```
if (typeof Object.create !== 'function') {
  Object.create = function(proto) {
    var F = function() {};
    F.prototype = proto;
    return new F();
  };
}

var base = {
  sayHowdy : function() { println("Howdy") }
}
var derived = Object.create(base)
for (var m in derived) {
  println(m + "=" + derived[m])
}
derived.sayHowdy()
```

Objects

This kind of "open object" system is extremely powerful programming

- very Lisp-ish/CLOS-ish in nature
- sometimes also known as Meta-Object Protocol (MOP)
- often used as building block for more powerful coding

Objects

Monkey-patching

```
// Method to add a method to any particular prototype
Function.prototype.method = function (name, func) {
  if (!this.prototype[name]) {
    this.prototype[name] = func;
  }
  return this;
};
// add an 'roundOff' method to Number
Number.method('roundOff', function() {
  return Math[this < 0 ? 'ceil' : 'floor'](this);
});
println((5.2).roundOff())
println((-12.2).roundOff())
```

Objects

Adding event-processing to any object

```
var eventuality = function(that) {
  var registry = {};
  that.fire = function(event) {
    var array, func, handler, i;
    var type = typeof event === 'string' ?
                 event : event.type;
    if (registry.hasOwnProperty(type)) {
      array = registry[type];
      for (i = 0; i < array.length; i++) {
        handler = array[i];
        func = handler.method;
        if (typeof func === 'string') {
          func = this[func];
        }
        func.apply(this, handler.parameters || [event]);
      }
    }
    return this;
  };
```

Objects

Adding event-processing to any object

```
  that.on = function(type, method, parameters) {
    var handler = {
      method : method,
      parameters : parameters
    };
    if (registry.hasOwnProperty(type)) {
      registry[type].push(handler);
    } else {
      registry[type] = [handler];
    }
    return this;
  };
  return that;
};
```

Objects

Adding event-processing to any object

```
var stooge = {
  "first-name" : "Jerome",
  "last-name" : "Howard"
};
var eventedStooge = eventuality(Object.create(stooge));

eventedStooge.on('poke', function() {
  println("Oh, a wiseguy, eh?");
});

eventedStooge.fire("poke");
```

Closure

Closure

- referenced values remain around as long as function does

  the function "closes over" the reference variable (hence the name)

- the actual link isn't explicit or discoverable

  this provides opportunities to "hide" members from the object on which a function operates, to avoid pollution

Advanced

Modules: Use closures to encapsulate state and hide details

```
String.method('deentityify', function() {
  var entity = { quot : '"', lt : '<', gt: '>' };
  return function () {
    return this.replace( /&([^&;]+)/g,
      function(a, b) {
        var r = entity[b];
        return typeof r === 'string' ? r : a;
      });
  };
} ());
// last line invokes the function, which returns a
// function, which is then the parameter to 'method'
// and gets added to the String prototype
// the entity array is only built once across all invocations
//

var s = "&lt;html&gt;"
print(s.deentityify())
```

Advanced

Currying: create new functions out of old by partially-applying the parameters required

```
function add (lhs, rhs) {
  return lhs + rhs;
}
Function.method('curry', function() {
  var slice = Array.prototype.slice,
      args = slice.apply(arguments),
      that = this;
  return function () {
    return that.apply(null,
      args.concat(slice.apply(arguments)));
  };
});
var add1 = add.curry(1);
var results = add1(6); // produces 7
```

Advanced

Memoization: remember the results of previous computations, to avoid rework

```
var fibonacci = function(n) {
  return n < 2 ? n : fibonacci(n-1) + fibonacci(n-2);
}
for (var i = 0; i <= 10; i ++) {
  println("Fibo " + i + ": " + fibonacci(i));
}
// computes fibonacci(3) a LOT of times
```

Advanced

Memoization: remember the results of previous computations, to avoid rework

```
var fibonacci = function() {
  var memo = [0, 1];
  var fib = function(n) {
    var result = memo[n];
    if (typeof result !== 'number') {
      result = fib(n - 1) + fib(n - 2);
      memo[n] = result;
    }
    return result;
  };
  return fib;
}();
for (var i = 0; i <= 10; i ++) {
  println("Fibo " + i + ": " + fibonacci(i));
}
// computes fibonacci(3) exactly once
```

Advanced

Memoization: generalization for all objects

```
var memoizer = function(memo, fundamental) {
  var shell = function(n) {
    var result = memo[n];
    if (typeof result !== 'number') {
      result = fundamental(shell, n);
      memo[n] = result;
    }
    return result;
  };
  return shell;
};

fibonacci = memoizer([0,1], function(shell, n) {
  return shell(n - 1) + shell(n - 2);
});
factorial = memoizer([1, 1], function(shell, n) {
  return n * shell(n - 1);
});
```

Forward

- First there was E4X...
  - it added XML literals and limited XPath refinement
  - but got no traction from the various players
- ... then there was ES4...
  - it added a LOT of stuff: namespaces, ...
  - but got limited traction from only a few players
- ... and don't forget ActionScript 3.0...
  - based on and prototyped around ES4, for Flash dev
  - but was only supported by Adobe
- ... now we have "Harmony"
  - smaller subset of ES4 that all players now agree on

Forward

- ES5 was approved in 2009
  - but most of it is just refinement of ES3
- For now, just stick with ES 3 features
  - as much as we'd like to believe that all the relevant players are now "on board" with the new standard, there was a time not that long ago when we believed that all the relevant players were "on board" with the standard being created, and we see how that turned out

NodeJS Modules

Because NodeJS is an ecosystem

NPM

Node Package Manager

Modules

Node doesn't have everything "out of the box"

- in fact, there is less in the box than expected
- fortunately, the box comes with a "grow the box" tool

Modules

npm: Node Package Manager

- command-line tool to install/manage node packages
- full list of packages is at http://search.npmjs.org/

  WARNING: this is a huge list, some are good, some are crap
  Better to know what you want before hunting for it

Modules

npm commands:

- ls: list all installed packages
- install {source}: most commonly used

  installs either per-user or (-g, --global) globally
  globally will require admin rights
  most of the time, this pulls from the NPM registry
  also installs dependencies listed for that package
- update {package}: install newest version of package
- uninstall {package}: remove a package
- help {command}: HTML help file for {command}
- docs {package}: Open HTML docs on {package} (maybe)
- folders: Where will stuff be installed?

Modules

Some interesting modules to explore:

- Socket.io: websockets server/client
- Connect: "middleware" framework
- Express: Web framework
- Geddy: Web framework
- Jade: templating engine
- TowerJS: Web framework++ (includes Connect and Express)
- More--see https://github.com/joyent/node/wiki/modules

Summary

The slide you've been waiting for: The End!

Summary

NodeJS represents...

- a way for JavaScripters to work both client- and server-side
- a new ecosystem that is pulling from the Ruby community
- some serious duplication of effort with ASP.NET MVC
- some easier (?) access to non-MSFT tools and systems
- hip and cool, and really, what other justification do you need?

References

Resources for Node

- NodeJS sources: http://nodejs.org
- NodeJS for Windows: http://go.microsoft.com/?linkid=9784334
- NodeJS Windows binaries: http://node-js.prcn.co.cc/
- iisnode: https://github.com/tjanczuk/iisnode
- NodeJS Azure SDK:
- NodeJS modules: https://github.com/joyent/node/wiki/modules
- Express: http://expressjs.com/
- "Node is not single-threaded" by Rick Garibay

  http://rickgaribay.net/archive/2012/01/28/node-is-not-single-threaded.aspx

Questions

?