Intro to Node.js

version 2.0, March 2019
www.watzthis.com

Introduction

Objectives

- Who am I?
- Who are you?
- Daily Schedule
- Course Schedule and Syllabus

About Me

3

Introductions

- What's your name?
- What do you do?
- JavaScript level (beginner, intermediate, advanced)?
- What do you want to know at the end of this course?
- Favorite food?

4

Daily Schedule

- 09:00 - 10:30
- 15 minute break
- 10:45 - 12:00
- 1 hour lunch break
- 1:00 - 2:00
- 15 minute break
- 2:15 - 3:15
- 15 minute break
- 3:30 – 5:00

5

What is Node.js?

- JavaScript runtime
- Runs JavaScript on the server (outside of the browser)
- Built on Google Chrome's V8 JavaScript engine
- Open-source

6

What is it good for?

- Web servers
- Networking tools
- Client-side tools

7

What is it not good for?

- Extremely computationally-heavy (CPU-bound) tasks
  - i.e. video encoding

8

History of Node.js

- 2009: Created by Ryan Dahl @ Joyent and introduced at JSConf
- 2010: Node's package manager, npm, was introduced
- 2012: Dahl stepped aside as project lead
- 2014: Node.js was forked to create io.js as an open governance alternative to Node.js
- 2015: Node.js Foundation was created and Node.js and io.js were merged back together.

9

Who Uses Node?

- Netflix
- NYTimes
- Paypal
- Uber
- Medium
- LinkedIn
- Walmart
- CBS
- eBay
- Pinterest
- HP
- Groupon
- Wall Street Journal
- Lowe's
- Capital One
- Microsoft
- Intel
- Amazon
- Github
- Google
- YouTube
- IBM
- etc etc etc

10

How Does Node.js Work?

- Event-driven
  - Listens for events and does things in response.
- Doesn't wait
  - If there's no event, Node is "sleeping."
  - If there's nothing left to do, the Node process exits

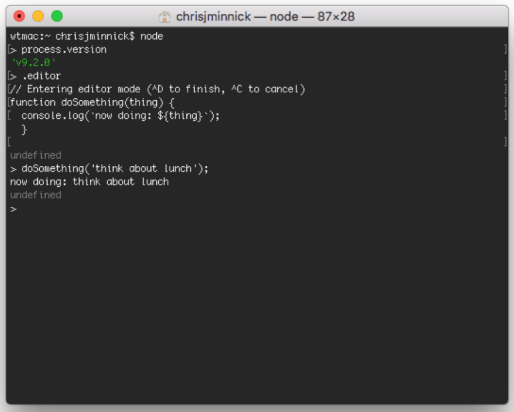
11

Node's REPL

- Read-Eval-Print-Loop
- Start by entering `node` in the command line without a program name.
- Accepts individual lines of user input, evaluates, then outputs results.
- Supports running node expressions and several special commands, including
  - .exit - exit REPL
  - .save - save the current REPL to a file
  - .load - load a file into the current session
  - .editor - enter editor mode

12

REPL Demo



13

Your First Node Program

Create a new file with a .js extension

- `vi my-first-program.js`

Write a statement in the new file to write to the console

- `console.log("Hello Node");`

Save and run the program from the command prompt

- `node my-first-program.js`

14

Running a Node.js program

- Type node followed by the program name.

```
node program.js
```

15

Lab 01: Intro to Node

- In this lab, you'll install Node.js, learn to use the interactive shell (aka REPL), and write a node application.

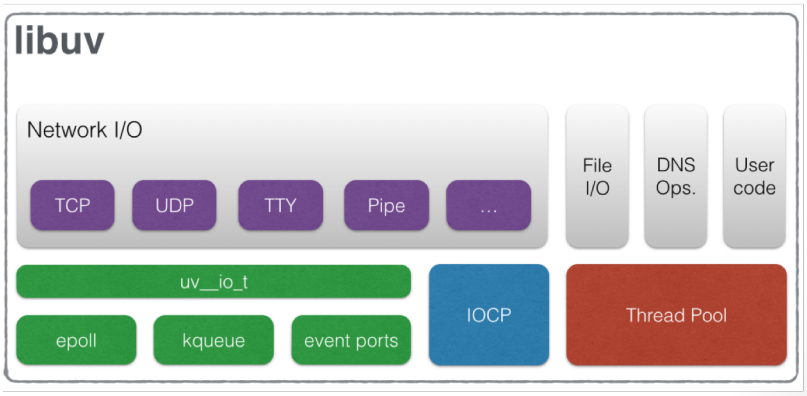
16

What is Node.js Made Of?

1. libuv
   - evented I/O library
2. V8
   - Google's JavaScript engine
3. Custom JS & C++ code

17

libuv - cross-platform I/O library



18

libuv's threadpool

- Used internally to run all file system operations
- Global and shared across all event loops
- libuv preallocates and initializes the max number of threads allowed
  - UV_THREADPOOL_SIZE
  - Overhead: ~1MB for 128 threads

19

Event loop

- Problem: JavaScript is single-threaded.
- The event loop allows Node.js to perform non-blocking I/O operations.
  - Offloads operations to the system kernel when possible
- Node initializes the event loop on startup
- Loops through 6 phases
- Between each run of the loop, Node.js checks if it's still waiting for anything and shuts down if not.

20

How is Node.js Different?

- JavaScript is single-threaded
- A Node application has one event loop, and gives you the benefit of multithreading with async operations
  - Can handle thousands of concurrent connections with minimal overhead
- No buffering
  - Node applications never buffer data.
  - Data is output in chunks.
- Non-blocking
  - Node doesn't wait for data to be returned from APIs.
  - While it's possible to write blocking code in Node.js, it's discouraged.

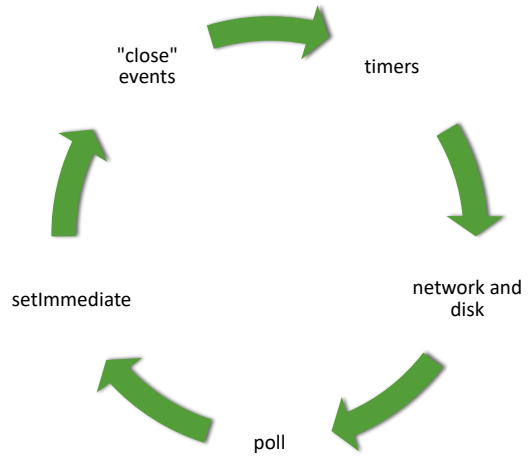
21

Event Loop Phases

1. timers - executes setTimeout() and setInterval() callbacks
2. I/O callbacks - executes most other callbacks
   - except close callbacks and setImmediate()
3. idle, prepare - internal use
4. poll - retrieve new I/O events
5. check - setImmediate() callbacks are executed
6. close callbacks - for example socket.on('close', ...)

22

Event Loop



process.nextTick()

- Not part of the event loop
- Adds a callback to the `nextTickQueue`
- `nextTickQueue` is processed after the current operation completes, regardless of the current phase of the event loop.
- Recursive `process.nextTick()` calls allows you to "starve" the I/O by preventing the event loop from reaching the poll phase.

Blocking vs. Non-blocking

Blocking

- In blocking code, the next statement doesn't execute until the previous one finishes.

```
var contents =
fs.readFileSync('file.txt','
utf8');
/* this line is reached when
the file is completely read
*/
console.log(contents);
```

Non-blocking

- In non-blocking code, execution doesn't wait.

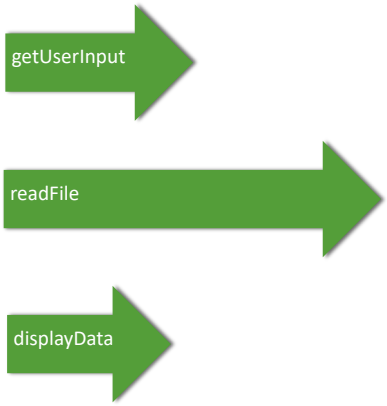
```
fs.readFile('file.txt',
'utf8', (err, data) => {
  /* this line is reached
     later when the results
     are in */
  console.log(data)
});
/* readFile returns
immediately and this line
is reached right away */
```

25

Blocking code

26

Non-Blocking code



27

Lab 02: First Look at Async Code

```
console.log('starting...');

process.nextTick(function(){
    for (var i=1; i<11; i++){
        console.log(i);
    }
});

console.log('Done!');
```

28

NODE MODULES

Modules Overview

- Node uses CommonJS for loading modules
- Original name was ServerJS
- Provides a way to import dependencies in JavaScript when used outside the browser (such as server side or in desktop applications)
- `module.exports`
  - Encapsulates code into a module
- `require`
  - Imports a module into JavaScript code

CommonJS Example

foo.js

```
var circle = require('./circle.js');
console.log('Area of circle: ' + circle.area(4));
```

circle.js

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


Using Modules

- Modules explicitly make their functionality available by exporting their functionality.
- Use `require()` to load a module and assign it to a variable.
  - `var foo = require("foo");`
- Modules can export variables, functions, or objects.
- If a module is a constructor function or class, import the module to a variable starting with a capital letter.
  - `var Bar = require('bar');`
  - `var myBar = new Bar();`
- If a module exports variables and functions, assign it to a camelCased variable.
  - `var foo = require('foo');`

Modules vs. Packages

- Modules are libraries for node.js
  - One-to-one file – module correspondence
  - Examples of modules:
    - Circle.js
    - Rectangle.js
    - Square.js
- Packages are one or more modules grouped together
  - Shapes ←Package name
    - Circle.js ← Modules that belong to the Shapes package
    - Rectangle.js
    - Square.js

33

Sources of Modules

- Node's core library
- Your project's files
- npm

34

npm

- Installs, publishes, and manages node programs
- Is bundled and installed with Node
- Allows you to install Node.js applications from the npm registry
- Written in JavaScript

35

node_modules

- Two ways to install npm packages:
  - Locally
  - Globally
- When you install packages locally, they're put into the **node_modules** directory in your current directory
- You should always run npm install from the same directory as your **package.json** file, which should be at the root of your project
- Run `npm update` to update local packages
- Run `npm outdated` to find out which packages are outdated

36

package.json

- Manages locally installed npm packages
- Documents packages your project depends on
- Specifies the versions of each package your project can use
- Makes your build reproducible
- Package versions are specified using Semantic versioning (`semver`)
- Semver ranges:
  - `~` : patch release - 1.0.x
  - `^` : minor release - 1.x
  - `*` : major release - x

37

Npm Install

- Is used to download and install a package
- `-g`
  - Install globally
- `--save`
  - Package will appear in your dependencies
- `--save-dev`
  - Package will appear in your `devDependencies`
- `--save-optional`
  - Package will appear in your `optionalDependencies`
- `--save-exact`
  - Saved dependency will be configured with an exact version rather than the default range operator

38

Examples of packages from npm

- lodash
  - a JavaScript utility library
- moment
  - parse, validate, manipulate, and display dates
- express
  - web application framework
- body-parser
  - body parsing middleware
- mocha
  - test framework
- xml2js
  - XML to JavaScript converter

39

Lab 03: Intro to npm

- In this lab, you'll learn about using npm for package management.

40

Node's Core Modules

- Certain modules are built into node and installed at the same time as Node.js.
- These modules can automatically be used in any application.
- Examples:
  - assert – used for writing util tests
  - crypto – provides cryptographic functionality
  - fs – used for performing file system operations
  - http – http server and client functionality
  - os – operating system-related utility methods

41

The http Module

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

- Core module for working with HTTP protocol
- Important Methods
  - `http.createServer()`
    - method for creating an instance of `http.Server`
  - `http.request()`
    - method for performing HTTP requests
  - `http.get()`
    - shortcut (convenience) method for GET requests

42

The http Module (cont)

- `http.Server`
  - class used for creating an HTTP server
- Important methods
  - server.listen()
    - starts the HTTP server listening for connections
    - first optional parameter is port number
      - server.listen(3000) will tell the server to listen on port 3000
  - server.close()
    - stops the server from accepting new connections

Lab 04: A Simple Node.js Server

- In this lab, you'll use Node.js and the http module to create a simple web server that listens for connections and responds with a simple Hello World message.

The `fs` module

- `var fs = require('fs');`
  - Performs filesystem operations
  - Contains blocking (a.k.a. synchronous) as well as non-blocking (asynchronous) methods
- Async example:

```
fs.readFile('hello.txt', function(err){
  if (err) throw err;
  console.log('success');
}
```

- Sync example:

```
fs.readFileSync('hello.txt');
console.log('success');
```

45

Buffer Objects

- Makes it possible to read and manipulate streams of binary data.
- Converts Buffer objects to strings like this:
  - `var str = buf.toString();`
- `fs` module returns a buffer, unless you pass 'utf8' as the 2nd argument

46

Modularizing Your Code

- Create a separate `.js` file
- Default export
  - Uses `module.exports = function(){...}` to export a single function or variable.
- Named export
  - Used for exporting multiple values from a file.
    - `module.exports.pi = 3.14;`
    - `module.exports.c = 299792458;`

47

Returning Values from Modules

- To return a value from a module, take a callback function as an argument and call that function with `err` or return data

```
module.exports =
function(firstNum,secondNum,callback){
  var sum = firstNum + secondNum;
  if (isNaN(sum)){
    callback("error: sum not a number");
  }
  callback(null,sum);
});
```

48

Using a Local Module

- Use local modules the same way as you use core modules
- Local modules must include the path to the file.
- `.js` is implied and can be left off

```
var sumModule = require('./sumModule');
sumModule(1,3,function(err,data){
  if(err) throw err;
  console.log("The sum is: " + data);
}
```

49

Lab 05: Creating Modules

- In this lab, you'll write your first node module and then use that module in a program.

50

ES6 Modules

- Starting with ES6, JavaScript supports modules natively
- Starting with v8.5.0, Node supports ES6 modules natively.
- Name ES6 modules with .mjs (sometimes known as the "Michael Jackson Solution") to tell Node to treat them as ES6 modules.
- Run Node with --experimental-modules flag to enable support for ES6 modules
  - `node --experimental-modules index.mjs`
- You can't use `require()` within ES6 modules

51

ES6 Modules

- 2 Types
  - named exports
    - multiple per module
  - default exports
    - 1 per module

- Named export
- lib.js

```
export function square(x) {
  return x * x;
}
```

- main.js

```
import {square} from 'lib';
```

- Default export
- myFunc.js

```
export default function() {
…
};
```

- main.js

```
import myFunc from 'myFunc';
```

52

EVENTS AND STREAMS

Non-blocking with Events

- Events can be used to write non-blocking code.
- Publish, Subscribe Design Pattern
  - An object can publish events
  - Other objects can subscribe to events and be notified when they happen.

Events

- Many objects emit events
- All objects that emit events are instances of process.EventEmitter

55

EventEmitter

- `EventEmitter` is a class that lets you listen for events and assign actions when those events occur
- Very loose way of coupling different parts of your application
- To register listeners:
  1. Create an instance of EventEmitter

```
const EventEmitter =
require('events').EventEmitter;
var eventEmitter = new EventEmitter();
```

  2. Register listeners with the eventEmitter.on() method.

```
eventEmitter.on('some-event',function(){
  // do something here
}
```

56

EventEmitter – emit events

- eventEmitter.emit() passes an arbitrary set of arguments to the listener functions.
- In other words, it emits events.

```
eventEmitter.on('sandwich', (quan, type)=>{
  console.log(`${quan} ${type} sandwich coming up!`);
});

eventEmitter.emit('sandwich', 1, 'Roast Beef');
```

57

EventEmitter Patterns – Pattern 1

- Extend EventEmitter
  - Create a module that inherits EventEmitter
  - Emit events from the module
  - Listen for events

58

Code Walkthrough: Extending EventEmitter

```
const EventEmitter = require('events');

class Resource extends EventEmitter {
  constructor(maxEvents){
    super();
    this.maxEvents = maxEvents;
  }
  process.nextTick(()=>{
        var count = 0;
        self.emit('start');
        var t = setInterval(function() {
            self.emit('data', ++count);
            if (count === maxEvents) {
                self.emit('end', count);
                clearInterval(t);
            }
        },10);
    });
};
modules.exports = Resource;
```


Code Walkthrough: Extending EventEmitter (cont)

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

  var r = new Resource(7);

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

  r.on('data', function(d) {
    console.log(`Got data: ${d}`);
  });

  r.on('end', function(t) {
    console.log(`Finished. Got ${t} events.`);
  });

EventEmitter Patterns - Pattern 2

- Return events from a function
  - Run a function that emits events
  - Listen for events the function can emit

61

Code Walkthrough: Emit events from a function

```
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(`Got data: ${d}`);
});

r.on('end', function(t) {
    console.log(`Finished. Got ${t} events.`);
});
```

62

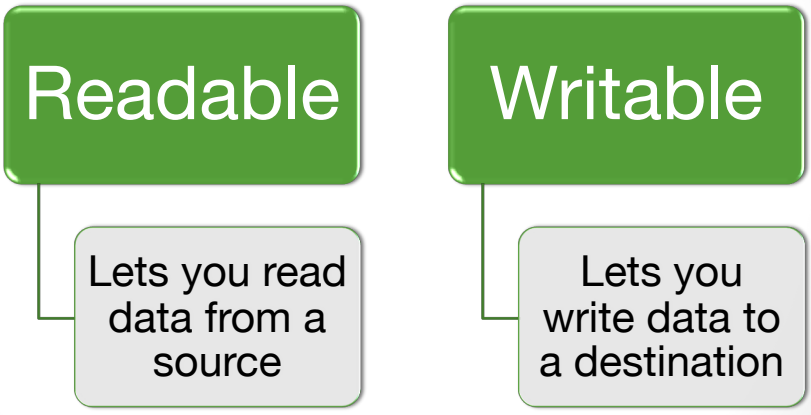
Node Stream Objects

- Unix Pipes for moving data around
- Objects that emit events
- Stream events
  - data
    - Emitted when a chunk of data is available
    - Chunk size depends on the data source
  - error
    - Emitted on error
  - end
    - Emitted when there's no more data to be consumed

```
response.on("data"), function(data){
  /* do something with data */
}
```

63

Types of Streams



64

Using Readable Streams

```
var fs = require('fs');
var readableStream =
fs.createReadStream('file.txt');
var data = '';

readableStream.on('data', function(chunk) {
    data+=chunk;
});

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

65

Using Writable Streams

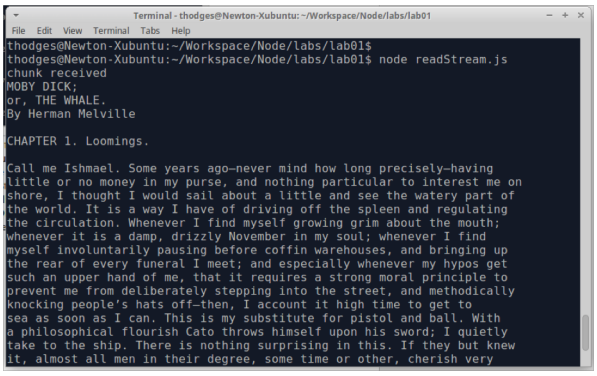
```
var fs = require('fs');
var readableStream =
fs.createReadStream('file1.txt');
var writableStream =
fs.createWriteStream('file2.txt');

readableStream.setEncoding('utf8');

readableStream.on('data', function(chunk) {
    writableStream.write(chunk);
});
```

66

Other Streams

- Duplex Streams
  - Both readable and writable
- Transform Stream
  - A duplex stream that modifies the data

Lab 06: Working with Streams



The pipe method

- A simpler method for consuming streams
- Pipes the output of a readable stream to a writable stream
- Automatically manages errors, end-of-files, and cases where one stream is faster than the other
- Example

```
readableStream.pipe(transformStream)
              .pipe(transformStream2)
              .pipe(destination)
```

- pipe() returns the destination, which enables chaining.

69

Lab 07: Piping Between Streams

70

The `process` Object

- The process object is a global that provides information about and control over the current Node.js process

71

Command Line Arguments

- You can pass command-line arguments to Node when you run a program
  - `node my-first-program.js 1 2 3`
- Command line arguments can be accessed using the process object
- Arguments are stored in the `process.argv` array
- First element of process.argv is 'node'
- Second element of proess.argv is the path to your program
- Your arguments come after that
- Example:
  - `[ 'node', '/path/to/my-first-program.js', '1', '2', '3' ]`

72

Command Line Arguments (cont)

- All elements of `process.argv` are strings
- "Number strings" must be converted to number data type prior to using for math
  - Two ways
    - `Number(process.argv[1])`
    - `+process.argv[1]`
- To loop through arguments in `process.argv`

```
for (i=2; i<process.argv.length; i++) {
  console.log('argument #'+ i + ': ' +
process.argv[i]);
}
```


Lab 08: The process object

Understanding Callbacks

- JavaScript functions are first-class objects
- They can be passed as arguments to other functions
- The other function can then execute the function passed to it.
- Also known as "Higher Order Function"

75

Using Callbacks

```
var myCallback = function(data) {
  console.log('got data: '+data);
};

var useData = function(callback) {
  callback('here is the data');
}

useData(myCallback);

output:
// got data: here is the data
```

76

Node's Error Convention

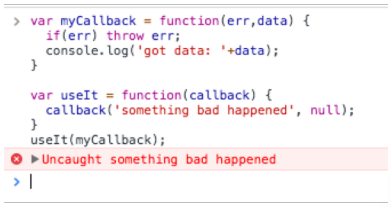
- The callback should expect to receive at least one argument.
- The first argument is an error.

```
var myCallback = function(err,data) {
  if(err) throw err;
  console.log('got data: '+data);
}

var useIt = function(callback) {
  callback(null, 'here is some data');
}
useIt(myCallback);
```

77

Node's Error Convention

```
var myCallback =
function(err,data)
{
  if(err) throw
err;
  console.log('got
data: '+data);
}

var useIt =
function(callback)
{

callback('something
bad happened');
}
useIt(myCallback);
```



78

Using Node's Error Convention

```
fs.readFile('file.txt', utf-8',
      function(err,content){
            if (err) {
                  return console.log(err);
            }

            console.log(content);
      }
);
```

- Can you explain what's happening here?

PROMISES

What Are Promises?

- An abstraction for asynchronous programming
- Alternative to callbacks
- A promise represents the result of an async operation
- Is in one of three states
  - pending – the initial state of a promise
  - fulfilled – represents a successful operation
  - rejected – represents a failed operation

81

Promises vs. Event Listeners

- Event listeners are useful for things that can happen multiple times to a single object.
- A promise can only succeed or fail once.
- If a promise has succeeded or failed, you can react to it at any time.
  - `readJSON(filename).then(success,failure);`

82

Why Use Promises?

- Chain them together to transform values or run additional async actions
- Cleaner code
  - Avoid problems associated with multiple callbacks
    - Callback Hell
    - Christmas Tree
    - Tower of Babel
    - Etc

Demo: Callback vs. Promise

- Callback
  - ```
    fs.readFile('text.txt',(err, file)=>{
        if (err){
            //handle error
        } else {
            console.log(file.toString());
        }
    });
    ```
- Promise
  - ```
    readText('text.txt')
        .then((data)=>{
                console.log(data.toString());
            }, console.error
        );
    ```

Using Promises

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

function readFileAsync (file, encoding) {
    return new Promise((resolve, reject)=>{
        fs.readFile(file, encoding, (err, data)=>{
            if (err) return reject(err);
            resolve(data);
        })
    })
}

readFileAsync('myfile.txt')
       .then(console.log, console.error);
```


Promises with Bluebird

```
var Promise = require('bluebird');
var fs = Promise.promisifyAll(require('fs'));

fs.readFileAsync("name", "utf8").then((data)=>{
    console.log(data.toString());
});
```

async/await

- Async function declarations return an AsyncFunction object.
- Async Functions can be halted (using await)
- They return a Promise
- Simplifies the behavior of using promises synchronously.

87

Upgrading to async/await

- Promises

```
const task = () => {
  return functionA()
  .then((valueA) => functionB(valueA))
  .then ((valueB) => functionC(valueB))
  .catch((err) => logger.error(err))
}
```

- async/await

```
const task = async() => {
  const valueA = await functionA();
  const valueB = await functionB();
  } catch (err) {
    logger.error(err);
  }
}
```

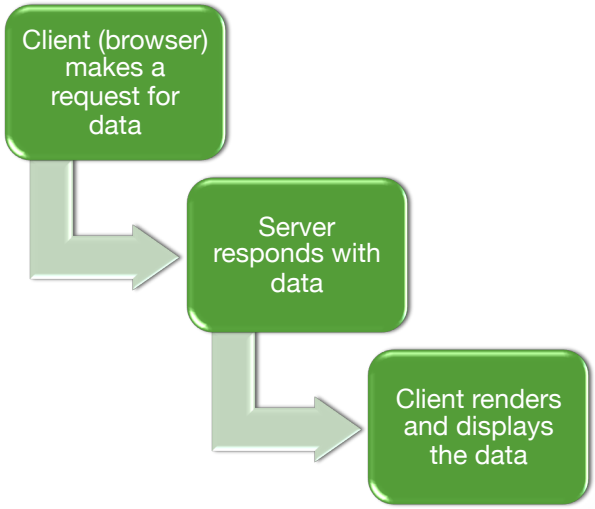
88

Lab 09: Promises

- Part 1: Asynchronous Callbacks Demo
- Part 2: Promises
- Challenges (optional)

NODE ON THE WEB

How the Web Works



HTTP

- Hypertext Transfer Protocol
- Runs on top of TCP/IP (Application level)
- Used for exchanging files on the web

HTTP Methods

- **GET**
- **POST**
- HEAD
- OPTIONS
- **PUT**
- **DELETE**
- TRACE
- CONNECT
- PATCH

93

Making an HTTP Request

- `http.get()`
  - First argument is the URL you want to GET
  - Second argument is a callback with the following signature:

```
function callback(response){ /* ... */ }
```

- The `response` object is a Node Stream

```
http.get('http://example.com/users/',function(response){
  response.on("data", function(data) { /* ... */ }
}
```

94

Lab 10: Getting Data with HTTP

- In this module, you'll use the http module to do an HTTP get request to a server. You'll then use the response stream to log each chunk of data from the server to the console.

95

RESTful Web Services

- REpresentational State Transfer
- Conforms to REST constraints, including:
  - Client-Server
    - separation of concerns
  - Stateless
    - All session information is kept on the client
  - Cache
    - responses from server must be marked as cacheable or non-cacheable
  - Uniform Interface
    - Each resource has its own URI
    - URIs don't change
    - Resources are manipulated through representations
      - Client doesn't interact directly with the server's resources
    - Each message must include enough information for the receiver to understand it in isolation.

96

RESTful Web Services in Practice

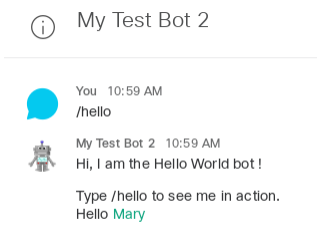
- Self-Descriptive Messages
  - HTTP methods follow their formal meaning
    - POST - create a new resource
    - DELETE - delete a resource
    - GET - retrieve a resource
    - PUT - update an existing resource
- JSON data

```
{
  "id":12,
  "firstname":"Han",
  "lastname":"Solo",
  "_links": {
    "self": {
      "href":"https://api.example.com/customers/12"
    },
    "orders": {
      "href":"https://api.example.com/customers/12/orders"
    }
  }
}
```

97

Lab 11: Making a Bot

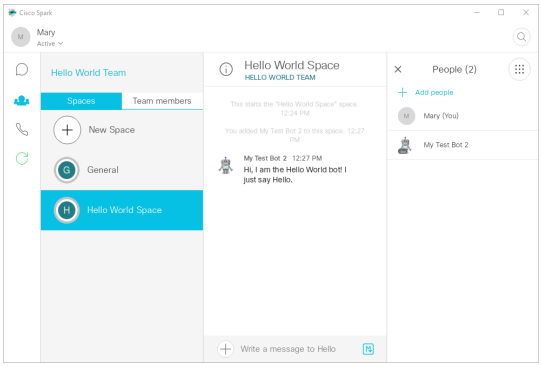
- In this lab, you'll create a bot for Cisco Webex Teams using sample code from CiscoDevNet. You'll use ngrok to allow the bot to communicate with Webex, and you'll configure Webhooks to notify your bot of events that occur on Webex.



98

Lab 12: Making a Hello World Bot

- In this lab, you'll program your first bot, which will just say hello when it's started up.



99

TESTING AND DEBUGGING

100

Test Driven Development

- Goal: Clean code that works.

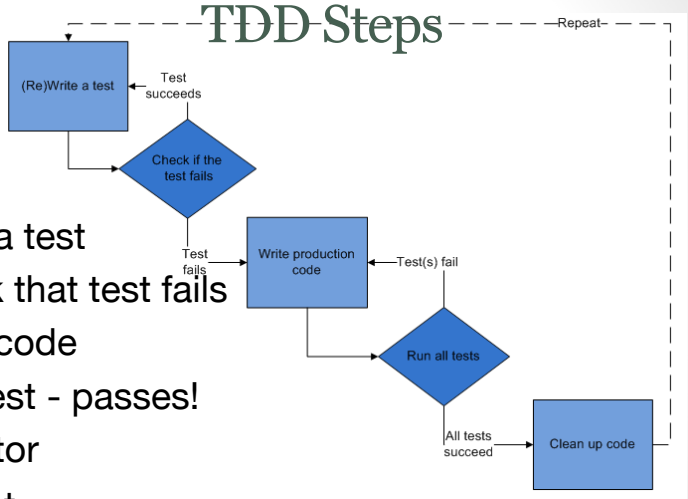
101

The TDD Cycle

- Red
  - write a little test that doesn't work.
- Green
  - make the test work, as quickly as possible.
  - don't worry about doing it right.
- Refactor
  - eliminate duplication created in making the test work.

102

TDD Steps



- Write a test
- Check that test fails
- Write code
- Run test - passes!
- Refactor
- Repeat

103

Red

- Write the story.
- Invent the interface you wish you had.
- Characteristics of a good tests:
  - Each test should be independent of the others.
  - Any behavior should be specified in only one test.
  - No unnecessary assertions
  - Test only one code unit at a time
  - Avoid unnecessary preconditions

104

Green

- Get the test to pass as quickly as possible.
- Three strategies:
  - Fake it
    - Do something, no matter how bad, to get the test to pass.
  - Use an obvious clean solution.
    - But don't try too hard!
  - Triangulation
    - only generalize code when you have two examples or more.
    - When the 2nd example demands a more general solution, then and only then do you generalize.

105

Refactor

- Make it right.
- Remove duplication.
- Improve the test.
- Repeat.
- Add ideas or things that aren't immediately needed to a todo list.

106

Assertions

- Expression that encapsulates testable logic
- Assertion Libraries
  - Chai, should.js, expect.js, better.assert
- Examples
  - `expect(buttonText).toEqual('Go!'); // jasmine`
  - `result.body.should.be.a('array'); // chai`
  - `equal($("h1").text(), "hello"); //QUnit`
  - `assert.deepEqual(obj1, obj2); //Assert`

107

JavaScript Testing Frameworks

- Jasmine
- Mocha
  - doesn't include its own assertion library
- QUnit
  - from JQuery
- js-test-driver
- YUI Test
- Sinon.JS
- Jest

108

JS Exception Handling

```
function hello(name) {
      return "Hello, " + name;
}
let result = hello("World");
let expected = "Hello, World!";
try {
      if (result !== expected) throw new Error
            ("Expected " + expected + " but got " +
            result);
} catch (err) {
      console.log(err);
}
```

109

TDD vs. BDD

Test-Driven Development

- Focused on being useful for programmers

Behavior-Driven Development

- Focused on documentation for non-programmers
  - Features, not results
  - More verbose

110

TDD Example

```
suite('Counter', function() {
  test('tick increases count to 1',

   function() {
       var counter = new Counter();
       counter.tick();
       assert.equal(counter.count, 1);
     });

});
```

111

BDD Example

```
describe('Counter', function() {
  it('should increase count by 1 after calling
tick',
   function() {
       var counter = new Counter();
       var expectedCount = counter.count + 1;
       counter.tick();
       assert.equal(counter.count, expectedCount);
     });
});
```

112

The assert module

- Contains assertions
  - equality
  - throws an exception
  - test for truthiness
  - test whether error parameter was passed to callback
  - more
- Types of equality
  - assert.equal(): shallow (==)
  - assert.strictEqual(); strict (===)
  - Date equality
  - Other object equality: equal if they have the same number of owned properties, equivalent values for each key, and the same prototype.

113

Using assert

```
var sumModule = require('./sumModule');
var assert = require('assert');

sumModule(1,2,(err,data)=>{
    assert.equal(data,3);
});

sumModule(1,'egg',(err,data)=>{
    assert.ifError(err);
}
```

114

JavaScript Frameworks and Assertion Libraries

- Frameworks
  - Jasmine
  - Mocha
  - AVA
  - Jest
  - Tape
  - Q
- Assertion Libraries
  - Assert
  - chai
  - should.js

Testing Asynchronous Code (with Mocha + should.js)

```
//sumModule.js
module.exports.sum = (number1, number2, callback) => {
    let sum = number1 + number2;
    if (isNaN(sum)) {
        callback("sum is not a number");
    } else if ((number1 <= 0) || (number2 <= 0)) {
        callback("all inputs must be positive")
    }
    callback(null,sum);
};

//sumModule.test.js
describe('sum', function() {
    it('should add numbers together', function(done){
        sumModule.sum(2,2,function(err,result) {
            result.should.equal(4);
            done();
        });
    });
});
```

Testing Synchronous Code (with Mocha + should.js)

```
//sumModule.js
module.exports.sumSync = (number1, number2) => {
    return number1 + number2;
};

//sumModule.test.js
describe('sumSync', function() {
    it('should add numbers together',
function(done){
        sumModule.sumSync(2,2).should.equal(4);
        done();
    });
});
```


Lab 13: Testing Node.js

- Run the tests in sumModuleTests.js
- Assert that the module will return an error if both arguments passed to it aren't positive.
- Run the test to confirm that it fails.
- Update sumModule to make the test pass.
- Convert the tests to use Mocha and Should.js

USING EXPRESS

What is Express?

- Web application framework for Node.js.

Getting Started with Express

1. Install Express
   - npm install express
2. Include Express in a file
3. Create an app object
   - var app = express();
4. Use app object methods
   - Routing
   - Configure middleware
   - Render HTML views
   - Register a template engine

Express Hello World Server

```
#app.js
const express = require('express');
const app = express();

app.get('/',(req, res)=>{
  res.send('Hello World!');
});

app.listen(3000,()=>{
  console.log('Listening on Port 3000');
});
```

Routing with Express

- Routing determines how an application responds to a client request to an endpoint.
- Endpoint
  - URI (path)
  - HTTP Request method
- Route definition structure
  - app.METHOD(PATH, HANDLER);
- METHOD
  - get, post, put, delete
- PATH
  - a path on the sever
- HANDLER
  - a function to execute when the route is matched.

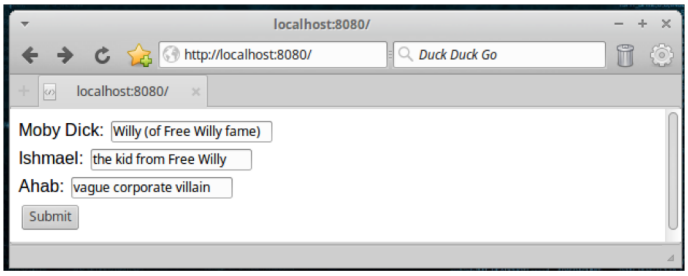
123

Serve Static Content with Express

- express.static
  - Built-in middleware function
  - Can be used to serve images, CSS, JavaScript, HTML
- app.use(express.static('public'))
  - Allows loading of files in the 'public' directory.
- Virtual path prefix
  - app.use('/static', express.static('public'))
    - Allows loading of assets from public directory using /static path
    - http://localhost:3000/static/images/tacos.gif

124

Lab 14: Make an Express Server



125

Database Access with Node.js

- Node can be used to access many different types of databases.
- The most popular DB to use with Node is MongoDB

126

mongod and mongo

- mongod is the server
- mongo is the command line

127

Lab 15: Working with MongoDB

128

API Testing with Mocha and Chai

Lab 17: Make Me hapi

- `cd labs/lab14-hapi16`
- `npm install`
- https://www.npmjs.com/package/makemehapi

Lab 18: Async Await and hapi 17

- `npm install --save hapi`
- hapi v17 is a major update
- Callbacks have been replaced with async await
- Open labs15/app.js
- Convert the basic server to hapi v17
- Convert the makemehapi exercises to hapi v17

JAVASCRIPT REVIEW

History of JavaScript

- Created in 1995 by Brendan Eich at Netscape
- Originally called Mocha, then LiveScript, then JavaScript
- Standardized as ECMAScript in 1997
- AJAX revolutionized the web in 2005
- In 2009, ECMAScript 3.1 (later ES5) became the accepted, unified, standard
- ES6 (ES2015) is the latest version

133

JavaScript is NOT Java

- They're as different as "Car" and "Carpet"
- The name was mostly for marketing

134

Is JavaScript "Real" Programming?

- Early versions of JavaScript were bad
- Early implementations (browsers) were bad
- Got a reputation early on
- With ES5+, JS has become a full object-oriented (OO) language
- Also resembles a functional language
- Most popular programming language today

Where Can You Use JavaScript?

Any web browser	Web server	Hardware
• Chrome • Firefox • IE	• Node.js	• Tessel • Arduino • Raspberry PI

How JavaScript Works

- Programmer writes JavaScript (human-readable) code
- Browser (or Node.js) loads JavaScript code
- JavaScript engine compiles and executes code
- Various methods improve performance
  - Just In Time (JIT) compiler
  - Directed Flow Graph
  - Concurrent compilation

Is JavaScript Slow?

"JavaScript trades performance for expressive power and dynamism." – Douglas Crockford

JavaScript used to be slow

Today's JavaScript engines can outperform Java

JavaScript Engine

- A program that executes JavaScript code.

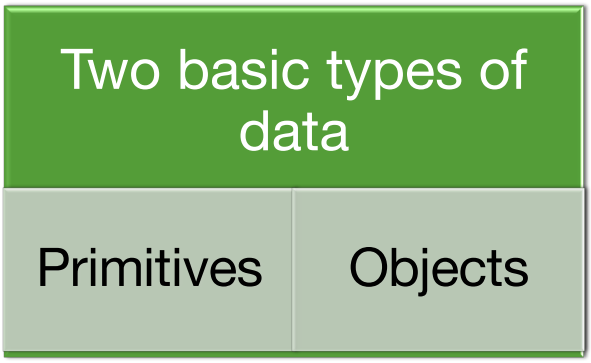
JavaScript Engine	Applications That Use It
V8	Chrome Node
Nitro (JavaScriptCore)	Safari
Chakra	Internet Explorer
Rhino	Firefox

JAVASCRIPT BASICS

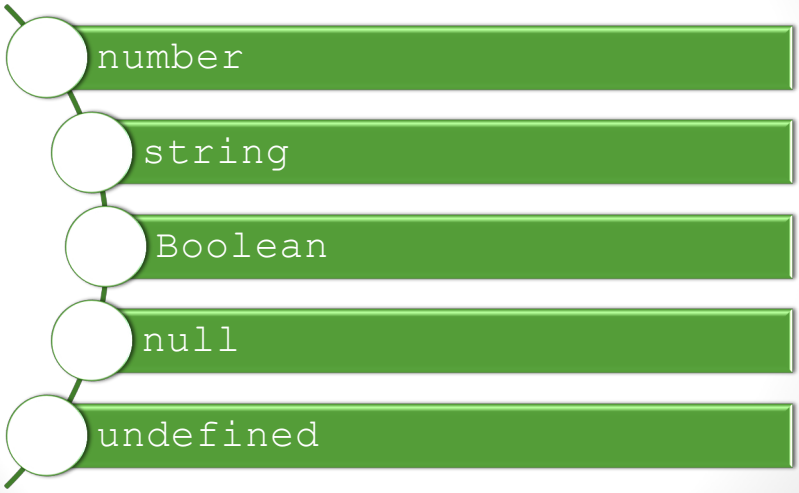
JavaScript Syntax

- Case-sensitive
- An *expression* produces a value
  - `1+1`
- A *statement* performs an action
  - `console.log(1+1);`
- Statements are separated by semicolons
- Strings can be enclosed by either single or double quotes.
- Comments can be block or line
  - `// line comment`
  - `/* block comment */`

JavaScript Data Types



JavaScript Primitives



143

Variables and Arrays

- Variables hold values
- Arrays are variables that can hold multiple values
- Any type of data can be stored in a variable or array
- Variable names must begin with a letter, underscore, or $
- Variable names are case-sensitive
- Certain reserved words can't be used as variable name

144

Creating and Using Variables

- To create a variable
  - `var myVariable;`
    - `creates a variable with function scope`
  - `let myVariable;`
    - `creates a variable with block scope`
  - `const myVariable`
    - `creates an immutable variable`
- To use a variable
  - `alert(myOtherVariable);`
  - `total = anotherVariable + 14`
- To change a variable's value
  - `myOtherVariable = "Something else";`
  - `anotherVariable = 0;`

145

Variable Scoping with `const` and `let`

- `const` creates constants
  - "immutable variables"
  - Cannot be reassigned new content
  - The assigned content isn't immutable, however,
    - If you assign an object to a constant, the object can still be changed.
- `let` creates block-scoped variables
  - Main difference between `let` and `var` is that the scope of `var` is the entire enclosing function.
  - Redeclaring a variable with `let` raises a syntax error
  - No hoisting
    - Referencing a variable in the block before the declaration results in a ReferenceError.

146

let vs. var

var

```
var a = 5;
var b = 10;

if (a===5) {
  var a = 4;
  var b = 1;
}
console.log(a); // 4
console.log(b); // 1
```

let

```
let a = 5;
let b = 10;

if (a===5) {
  let a = 4;
  let b = 1;
}
console.log(a); // 5
console.log(b); // 10
```


Creating and Using Arrays

- To create an array
  - `var myArray = []; // Creates an empty array`
  - `var myArray = ['thing1','thing2','thing3'];`
- To use an array
  - `alert(myArray[0]); // alerts 'thing1'`
  - `var myOtherArray[0] = myArray[3];`
- To change an array's value
  - `myArray[4] = 'thing5';`

JavaScript Operators

Operator	Name	Description	Example
=	assignment	sets the left equal to the right	`var a =3;`
+	addition	adds left and right	`var b = a + 1;`
-	subtraction	subtracts left from right	`var c = b - a;`
*	multiplication	multiples left and right	`var d = c * b;`
/	division	divides left by right	`var e = d / b;`
%	modulo	gets remainder of dividing left by right	`var f = d%2;`
++	increment	adds one	`var g = f++`
--	decrement	subtracts one	`var h = g--`

149

JavaScript Operators, cont.

Operator	Name	Description	Example
+	concatenation	combines two strings	`var i = "hi, " + "Jack!";`
==	equal	returns true if both sides are equal	`var j = a == b`
!=	not equal	returns true if both sides are not equal	`var k = a != b;`
===	strict equal	returns true if both sides are equal and are the same type	`var l = a === b;`
!==	strict not equal	returns true if both sides are not equal or if they're a different type	`var m = a !== b;`

150

JavaScript Operators, cont.

Operator	Name	Description	Example
>	greater than	returns true if the left is greater than the right	`var n = a > b`
<	less than	returns true if the left is less than the right	`var n = a < b`
>=	greater than or equal	returns true if the left is greater than or equal to the right	`var o = a >= b`
<=	less than or equal	returns true if the left is less than or equal to the right	`var p = a <= b`

Template Literals

- String Interpolation

```
var customer = { name: "Penny" }
var order = { price: 4, product: "parts", quantity: 6
}
message = `Hi, ${customer.name}. Thank you for your
order of ${order.quantity} ${order.product} at
${order.price}.`;
```

Functions

- Programs within programs
- Variables declared inside a function only exist in that function
- Invoke functions using their name, followed by parentheses
- Optional comma-separated parameters may be passed between parentheses
- Examples:
  - `var a = parseInt("10");`
  - `var b = sayHello("World");`
  - `var c = String(1000);`

153

Global Functions

- `decodeURI()`
- `decodeURIComponent()`
- `encodeURI()`
- `encodeURIComponent()`
- `escape()`
- `eval()`
- `isFinite()`
- `isNaN()`
- `Number()`
- `parseFloat()`
- `parseInt()`
- `String()`
- `unescape()`

154

Custom Functions

- Create functions inside your programs
- Use a `function` definition to create a function
- Use the `return` keyword to make a function return a value other than undefined

```
function myFunc(someValue) {
  var newValue = someValue + " is my value";
  return newValue;
}
```

- To invoke `myFunc()` and assign its result to a variable

```
var result = myFunc(33);
```

155

Arrow Functions

- aka "fat arrow" functions
- A more concise syntax for writing functions
  - ES5 (old way)

```
function increment(v){
   return v+1;
}
```

  - Arrow function

```
increment = (v) =>{v+1};
```

156

Arrow Function Parameters

- Surround parameter names with parentheses

```
(param1,param2,param3) => { statements }
```

- Parentheses are optional when there's only one parameter name

```
singleParam => { statements }
```


Arrow Functions (cont.)

- More intuitive handling of current object context.
  - ES6

```
this.nums.forEach((v) => {
  if (v % 5 === 0)
    this.fives.push(v);
});
```

  - ES5

```
var self = this;
this.nums.forEach(function (v) {
  if (v % 5 === 0)
    self.fives.push(v);
});
```

JavaScript Objects

An object is a collection of properties

A property is made up of a name and a value

A property may have a primitive value, an object value, or a function value

When a property has a function value, it's called a method

159

Built-in Objects

- JavaScript has built-in objects
- You can use methods and properties of built-in objects by using dot notation
- Examples of built-in objects
  - Array
  - Boolean
  - Function
  - Date
  - Error
  - Object
  - RegExp

160

Creating Objects

- You can create custom objects in your programs
- Three ways to create objects
  1. Object literal (aka object initializer)

```
var myCar = {color: "black", doors: 4}
```

  2. Constructor Function

```
function myCar(color,doors) {
  this.doors = doors;
  this.color = color;
}
```

  3. `Object.create`
     - Allows you to specify the object the new object is based on

```
var cat = Object.create(Animal);
```


Using Objects

- Two ways to access and change properties
  - Dot notation
    - `var howManyDoors = myCar.doors;`
    - `alert(myCar.doors);`
  - Square brackets
    - `var howManyDoors = myCar['doors'];`
  - Methods can be set and run the same way
    - `var myCar.start = function(key,ignition){return key + ignition;}`
    - `console.log(myCar.start("click","vroom"));`

Prototypal Inheritance

- JavaScript objects have a link to a prototype object
- You can create new objects by copying an existing one
- When you access a property on an object, JavaScript looks for that property in the prototype (and all the way up the chain) if it doesn't find it on the referenced object.