Node.js

Server side Javascript

Node.js

- Node.js is server side framework for javascript that runs on google’s chrome V8 engines.

- Node.js is single threaded event based server framework

- Comes with a lot of functionality “compiled” and bundled into the runtime beyond the basic JS spec

Modules

- All of Node’s functionality is bundled into modules and we have to require the modules to be able to use them.
- “fs” module contains a lot of operations related to the filesystem.
- Using the below functions, write a function that gets a list of files in a directory and another to make directory (http://nodejs.org/api/fs.html)

```
var fs = require(“fs");
fs.existsSync(path);
fs.statSync(path);
fs.readdirSync(path);
fs.mkdirSync(newDir);
```

Build a Command Processor

- Build a class called CommandProcessor
  - processCommand(cmdName, paramsArr)
  - Call the previously written methods from here based on the cmdName

Error Handling

```
try{
   throw "Something went wrong";
}catch(err){
   console.log(err);
}

try{
   throw new Error("Something went wrong");
}catch(err){
   console.log(err.stack);
}
```

Error Handling in Node

- When should I throw an error, and when should I emit it with a callback
- What should my functions assume about their arguments?
- How should I deal with arguments that don't match what the function expects? Should I throw an exception or emit an error to the callback?
- How can I provide enough detail with my errors so that callers can know what to do about them?
- How should I handle unexpected errors? Should I use try/catch, domains, or something else?

Types of Errors

- Operational Errors
- Programmer Bugs

Delivering Errors

- throw
- callback - Node.js convention callback(err, result)
- EventEmitter

Guidelines For Interfaces

- The documentation for every interface function should be very clear about:
  - what arguments it expects
  - the types of each of those arguments
  - any additional constraints on those arguments (e.g., must be an email address)
- If any of these are wrong or missing, that's a programmer error, and you should throw immediately.
- Document:
  - what operational errors callers should expect (including their names)
  - how to handle operational errors (e.g., will they be thrown, passed to the callback, emitted on an event emitter, etc.)
  - the return value

Delivering Errors

- The Error object is a built-in object that provides a standard set of useful information when an error occurs, such as a stack trace and the error message.
- Use Error objects (or subclasses) for all errors, and implement the Error contract.
  - You should provide name and message properties
- Wrap underlying errors and re-throw/send providing additional context (Use verror module)

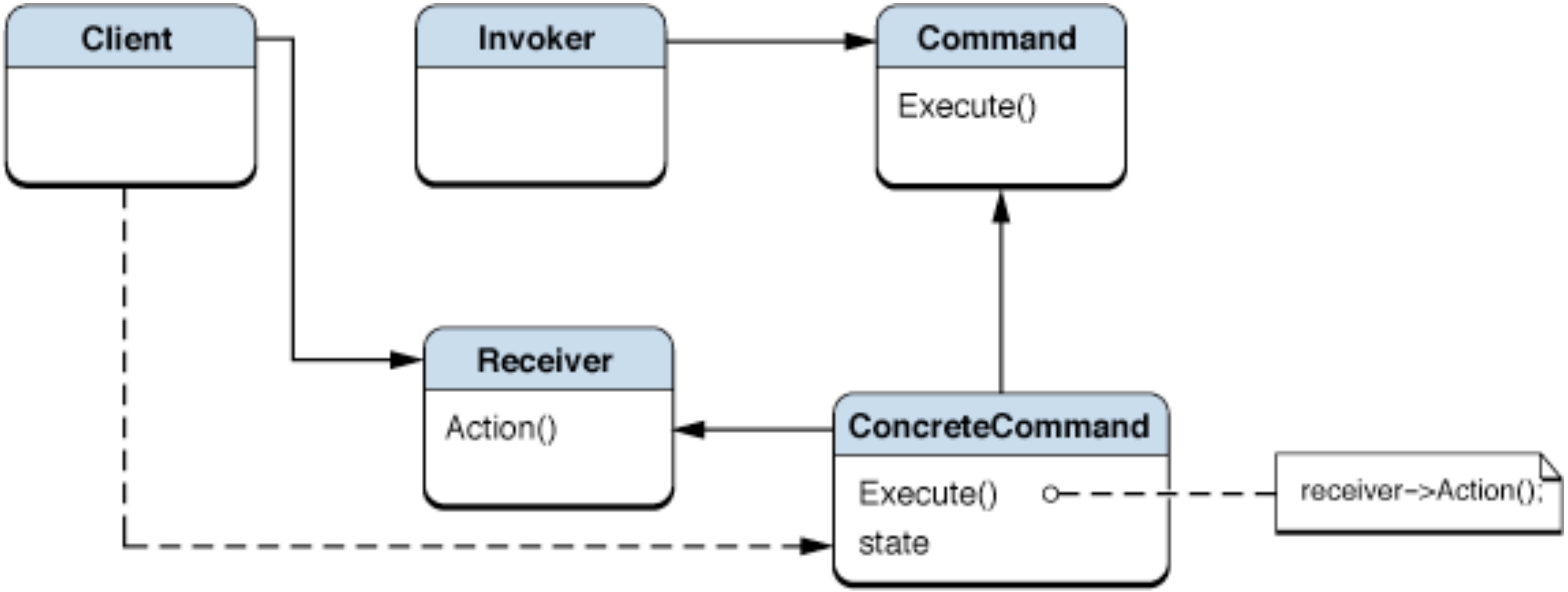
```
fs.stat(filename, function (err1) {
    var err2 = new VError(err1, 'stat "%s" failed', filename);
    console.error(err2.message);
});
```

Validate Params

- Validate the input parameters for the command functions to make sure they are the right number and type. ONLY if you are writing a library for thirdparty use
  - Throw errors upon validation failures

Implement Command Pattern

- Create each command in a class of its own and the base class invokes the necessary class based on command



For Command Pattern

- Implement CommandProcessor, ListCommandProcessor, MkdirCommandProcessor
- processCommand method of base class CommandProcessor only calls one of the child classes
- CommandProcessor (Base class) has a list of child class instances.

Make It Event-Driven

- Use the corresponding async versions of the methods

```
fs.exists
fs.stat
fs.readdir
```

- Change processCommand method to accept callbacks and send the errors back in callbacks

EventEmitters

- EventEmitter is a base class in Node.js
- emit() and on() methods to raise and handle events
- Can be used to handle errors
- Modify command processor to use event emitter to notify non exiting path on ls command

```
var EventEmitter = require(‘events').EventEmitter;
CommandProcessorAsyncE.prototype =
                                Object.create(EventEmitter.prototype);
this.emit("error", new Error("Specified path does not exist"));
cp.on("error", function(msg){

});
```

Buffers

- Raw data is stored in instances of the Buffer class.
- A Buffer is similar to an array of integers but corresponds to a raw memory allocation
- A Buffer cannot be resized.
- Converting between Buffers and JavaScript string objects requires an explicit encoding method.

Implement The “cat”

- Unix cat command displays contents of file
- Implement a new command processor subclass called CatCommandProcessor that will read the contents of a file and send to a callback
- Use these methods:

```
fs.open(path, "r", callback);
fs.read()
var buffer = new Buffer(1000);
```

Callback Sequencing

- When we fire multiple asynchronous methods, the callbacks do not necessarily come in the same order

- One way to fix this is to embed a sequence number into a callback using a closure

- Try numbering the async read calls into a sequence

Callback Sequence Solution

```
var  callBackFactory = function (sequence){
   var sequenceCount=sequence;
   return function(err, bytesRead, buffer){
      dataString+= buffer.toString("ascii");
      console.log(sequenceCount+ ". got data of size: "+bytesRead);
      if(bytesRead==1000){
         //There is probably more to read
         buffer.fill(32);//Fill spaces
         fs.read(fileDescriptor, buffer, 0, 1000, null,
                  callBackFactory(readSequence++));
      }else{
         //Done reading.. call the end callback
         callBack(null,dataString);
      }
   };
};
```

Node Streams

- A stream is an abstract interface implemented by various objects in Node.
- Types: Readable streams, Writable streams, Duplex streams, and Transform streams.
- Readable streams have two "modes": a flowing mode and a paused mode.
  - In paused mode, you must explicitly call stream.read() to get chunks of data out. Streams start out in paused mode.
- Streams are event emitters and same syntax with “on” is used

Stream Events

- Readable:
  - readable
  - data
  - end
  - close
  - error
- Writable:
  - drain
  - finish
  - pipe
  - unpipe
  - error

Piping

- piping streams helps channel data from a readable to writeable stream
- r.pipe(w)
    - Automatically throttles data based on drain events and data events internally
- Change the implementation of the cat command to use streams

```
var readStream = fs.createReadStream(path);
readStream.on(“data”,function(chunk){});
readStream.on(“end”,function(){});
```

Socket Streams

- net module can be used to create network aware applications using streams

- Here is an echo server

```
var net = require('net');
var server = net.createServer(function(socket) { // 'connection'
listener
   var dataStr = "";
   console.log('client connected');
   socket.write("Hello welcome to the echo server")
   socket.pipe(socket); //for echoing
});
server.listen(4000, function() { // 'listening' listener
   console.log('server listening');
});
```

Convert the Command Processor to work off telnet

```
var server = net.createServer(function(socket) { // 'connection'
   console.log('client connected');
   socket.on('data', function(buffer) {
      console.log('data arrived: '+buffer.toString('ascii'));
      //Process command and write response to socket
   });
   socket.write('hello, enter your command:\r\n');
});
server.listen(4000, function() { // 'listening' listener
   console.log('server bound');
});
```

Modularising Code

- Code can be moved to other files and exported from there
- Only exported entries will be visible outside

```
var foo = function() { ... }
var bar = function() { ... }
var blah = function() { ... }
exports.foo = foo;
exports.bar = bar;

var myMod = require('./my_module');
myMod.foo();
myMod.bar();
```

Regular Module Search

- For example, if the file at '/home/user/projects/foo.js' called require('bar.js'), then node would look in the following locations, in this order:
- /home/user/projects/node_modules/bar.js
- /home/user/node_modules/bar.js
- /home/node_modules/bar.js
- /node_modules/bar.js

Making Folders as Modules

- It is convenient to organise programs and libraries into self-contained directories, and then provide a single entry point to that library
  - place a package.json in the folder with this

```
{ "name" : "some-library",
  "main" : "./lib/some-library.js" }
```

  - Alternatively node will load index.js or index.node file from that directory

Move Command Processor

- Lets move command processor into a separate js file including all commands

```
module.exports.CommandProcessorAsyncE = CommandProcessorAsyncE;

__________

var cp = require('./CommandProcessor.js');

var cpasynce = new cp.CommandProcessorAsyncE();
```

WebSockets

- Its a protocol now standardised for full duplex communication
- Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request.
- WebSocket protocol client implementations try to detect if the user agent is configured to use a proxy when connecting to destination host and port and, if it is, uses HTTP CONNECT method to set up a persistent tunnel.

socket.io and socket.io-client

- Socket.io library supports websockets in node.js
- Allows for EventEmitter style communication between the client and the server
- They need a http listener but thats only to setup a handshake

socket.io client server

```
var http = require('http').Server();
var io = require('socket.io')(http);

io.on('connection', function(socket){
  console.log('a user connected');
  socket.on("clientevent",function(message){
       console.log("$$$ Clientevent: "+message.msg);
       socket.emit("serverevent","Thanks for the message");
  });

});

http.listen(8000, function(){
  console.log('listening on *:8000');
});
------------------------------------------------------------------
var socket = require('socket.io-client')('http://localhost:8000');
socket.on('connect', function() {
   console.log("#### connected to socket server");
   socket.emit("clientevent", {msg: "Hello there..."});
});

socket.on('serverevent', function(message) {
   console.log("#### Message from server: "+message);
});
```

Socket IO command processor

- Socket IO command processor allows the client to be asynchronous with the server

- Submit the command in an event and the result comes back from server in a different result event

- Modify the command processor to use asynchronous network IO

NodeUnit

- Unit testing framework for node
  - npm install nodeunit --save-dev -g
- nodeunit is designed on the lines of junit and helps async testing
- Has got a command line test runner

Basic Tests

- create a test.js with this content and launch it with nodeunit test.js on command line

```
exports.testModulePresense = function(test){
   try{
       var casync = new cp.CommandProcessorAsyncE();
       test.ok(true, "All ok");
   }catch(err){
      test.ok(false,"Error, could not create cp: "+err);
   }
    test.done();
};
```

Asserts

- ok(value, [message]) - Tests if value is a true value.

- equal(actual, expected, [message]) - Tests shallow, coercive equality with the equal comparison operator (==).

- deepEqual(actual, expected, [message]) - Tests for deep equality.

- strictEqual(actual, expected, [message]) - Tests strict equality, as determined by the strict equality operator (===)

- throws(block, [error], [message]) - Expects block to throw an error.

Testing Async

```
exports.testAsync = function(test){
   setTimeout(function(){
      test.ok(true,"All ok");
      test.done();
   },2000);
};
```

TestGroups & Setup

```
exports.test1 = function (test) {
    ...
}

exports.group = {
   setUp : function(callback) {
      callback();
   },
   tearDown : function(callback) {
      callback();
   },
    test2: function (test) {
        ...
    },
    test3: function (test) {
        ...
    }

}
```

Expecting Assertions

```
exports.testAsync = function(test){
   test.expect(2);
   setTimeout(function(){
      test.ok(true,"All ok");
      setTimeout(function(){
         test.ok(true,"All ok");
         test.done();
      },2000);
   },2000);
};
```

Mocking

```
setUp : function(callback) {
      console.log("Calling setup...");
      fs_exists = fs.exists;
      fs.exists = function(path,callback) {
         console.log("Exists called...");
         callback(true);
      };
      callback();
   },
   tearDown : function(callback) {
      console.log("Calling tear down...");
      fs.exists = fs_exists;
      callback();
   },
```

Lets Try It

- Test the cat and ls commands in two tests belonging to the same group

- Testing can be done from js aswell

```
var reporter = require('nodeunit').reporters.default;
reporter.run(['test.js']);
```

- Put this in package.json and test with npm test

```
"scripts": {
      "test": "nodeunit test.js"
  }
```

Node.js performance

- Node.js is single threaded, CPU intensive operations block the thread and reduce performance.
- Lets take a look at multi user performance of Node servers vis-a-vis traditional servers

Clustering

- The cluster module allows you to easily create child processes that all share server ports.
- We have the concept of cluster master and cluster workers.
- Master and workers communicate with each other using IPC
- Master can send messages to worker and vice versa.
    - Workers do not share any state

Forking for Workers

```
if (cluster.isMaster) {
   // Fork workers.
   for (var i = 0; i < cores; i++) {
      cluster.fork();
   }
} else {
   // Workers can share any TCP connection
   // In this case its a Socket server
   var server = net.createServer(function(socket) { // 'connection' listener
      socket.write(new Buffer("Hello World"));
      socket.end();
   });
   server.listen(5000, function() { // 'listening' listener
      console.log('server bound');
   });
}
```

Create a global Car object with model as property. If the car instance is empty, create a new instance and return the car’s model in socket write

Separating Master

```
var cluster = require('cluster');
var net = require("net");
var cores = require('os').cpus().length;
cluster.schedulingPolicy = cluster.SCHED_RR;
cluster.setupMaster({
    exec : 'clusteredApp.js',
});
for (var i = 0; i < cores; i++) {
    cluster.fork();
}
cluster.on('exit', function(worker, code, signal) {
    if (worker.suicide === true) {
        console.log('Ignoring suicide death');
    } else {
        // Spawn another process to replace the dead worker
        console.log("Replacing a dead worker!");
        cluster.fork();
    }
});
```

Grunt

- Grunt is a javascript task runner
- Is used as a build and automation tool to run code checking, minifying, testing and other activities
- Grunt has many pluggins to choose from for each of the tasks.
- Install grunt command line interface: npm install -g grunt-cli
    - This loads the local installation of the Grunt library, applies the configuration from your Gruntfile, and executes any tasks you've requested for it to run.

Gruntfile.js

- The Gruntfile.js file is a valid JavaScript that belongs in the root directory of your project, next to the package.json file
- A Gruntfile is comprised of the following parts:
  - The "wrapper" function
  - Project and task configuration
  - Loading Grunt plugins and tasks
  - Custom tasks

Hello World

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

grunt.registerTask('default', 'default task description',
function(){
  console.log('hello world');
});
```

Grunt Tasks With Params

- run this with grunt build:production or just grunt build

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

grunt.registerTask('default', 'default task description', function(){
  console.log('hello world');
});

grunt.registerTask('build', 'Performing build', function(name){
  if(!name || !name.length){
    grunt.warn('you need to provide an environment.');
    name = 'development';
  }

  console.log('building for: ' + name);
});
```

Chaining Tasks

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

grunt.registerTask('deploy', 'deployment task', function(){
  console.log('Deploying the app...');
});

grunt.registerTask('build', 'Performing build', function(name){
  if(!name || !name.length){
    grunt.warn('you need to provide an environment.');
    name = 'development';
  }

  console.log('building for: ' + name);
});

grunt.registerTask('default', ['build:development','deploy']);
```

Multi Tasks

- run as grunt buildAll or grunt buildAll:dev

```
grunt.initConfig({
   buildAll : {
      dev : {server: 'localhost'},
      prod : {server: '10.33.21.121'},
   }
});

grunt.registerMultiTask('buildAll', 'Building all', function() {
   grunt.log.writeln("Building for: "+this.target + ' on server: ' +
this.data.server);
});
```

Grunt plugins: nodeunit

```
grunt.initConfig({
   nodeunit: {
       all: ['test.js']
   }
});
grunt.loadNpmTasks(‘grunt-contrib-nodeunit');
```

run as: grunt nodeunit

Grunt plugins: jshint

```
grunt.initConfig({
   jshint: {
      all: ['*.js'],
      jshintrc: true
   }
});
```

```
grunt.loadNpmTasks(‘grunt-contrib-jshint');
```

run as: grunt jshint

Minifying Sample

```
module.exports = function(grunt) {
   // Project configuration.
   grunt
   .initConfig({
      pkg : grunt.file.readJSON('package.json'),
      uglify : {
         options : {
            banner : '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-
dd") %> */\n'
         },
         build : {
            src : '*.js',
            dest : 'build/<%= pkg.name %>.min.js'
         }
      }
   });
   // Load the plugin that provides the "uglify" task.
   grunt.loadNpmTasks('grunt-contrib-uglify');
   // Default task(s).
   grunt.registerTask('default', [ 'uglify' ]);
};
```

Promises

- Promises are objects that have a “then” method
- Unlike node functions, which take a single callback, the then method of a promise can take two callbacks: a success callback and an error callback.
    - When one of these two callbacks returns a value or throws an exception, then must behave in a way that enables stream-like chaining and simplified error handling.

Without Promises

```
function doSomething(callBack) {
    setTimeout(function() {
        console.log("2 seconds up");
        callBack(null, "2 seconds up");
    }, 2000);
}

function doSomethingElse(callback) {
    setTimeout(function() {
        console.log("2 more seconds up");
        callback(null, "2 more seconds up");
    }, 2000);
}

doSomething(function() {
    doSomethingElse(function() {
        console.log("Should be 4 seconds now!");
    });
});
```

With Promises

```
var Promise = require("bluebird");
function doSomethingWithPromise() {
    return new Promise(function(resolve, reject) {
        console.log("2 seconds up");
        resolve("2 seconds up");
    });
}

function doSomethingElseWithPromise() {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            console.log("2 more seconds up");
            resolve(null, "2 more seconds up");
        }, 2000);
    });
}

doSomethingWithPromise()
.then(function() {
    return doSomethingElseWithPromise();
})
.then(function() {
    console.log("Should be 4 seconds now!");
});
```

Promisification

```
var doSomethingWithPromise = Promise.promisify(doSomething);
var doSomethingElseWithPromise = Promise.promisify(doSomethingElse);


doSomethingWithPromise()
.then(function() {
   return doSomethingElseWithPromise();
}).then(function() {
   console.log("Should be 4 seconds now!");
});
```

Changes With Promises

```
//Regular Node code
fs.readFile(file, function(err, res) {
    if (err) handleError();
    doStuffWith(res);
});

//With promises
fs.readFile(file).then(function(res) {
    doStuffWith(res);
}, function(err) {
    handleError();
});
```

Promises...

- Promises are kept in future too

```
var filePromise = fs.readFile(file);
//do more stuff... and then
filePromise.then(function(res) { //process data in res
});
```

- Promises are good for multiple things

```
filePromise.then(function(res) { uploadData(url, res); });
filePromise.then(function(res) { saveLocal(url, res); });
```

- Returns from promises become promises

```
var firstLinePromise = filePromise.then(function(data) {
    return data.toString().split('\n')[0];
});

var beginsWithHelloPromise = firstLinePromise.then(function(line) {
    return /^hello/.test(line);
});
```

Error Handling With Promises

```
function readProcessAndSave(inPath, outPath) {
    var filePromise = fs.readFile(inPath);
    var transformedPromise = filePromise.then(function(content) {
        return service.transform(content);
    });
    var writeFilePromise =
           transformedPromise.then(function(transformed) {
        return fs.writeFile(otherPath, transformed)
    });
    return writeFilePromise;
}
readProcessAndSave(file, url, otherPath).then(function() {
    console.log("Success!");
}, function(err) {
    console.log("Error: ", err);
});
```

Promisify

- Bluebird.js is a promise library that allows for converting regular node libraries to support promises

- For this to work, node libraries should follow the convention of callback(err,successVal)

```
Promise.promisifyAll(CommandProcessorAsyncE.prototype);

//Method will be called in context of cpasyncce
var cp = Promise.promisify(cpasynce.processCommand, cpasynce);

var fs = require("fs");
Promise.promisifyAll(fs);
```

Chaining And Error Handling

```
return fs.readFile(inPath)
      .then(service.transform)
      .then(function(transformed) {
        return fs.writeFile(otherPath, transformed)
      }).caught(function(e){
        console.log("Error in operations");
        console.log(e.stack);
      }).lastly(function(){
        console.log("Executed finally");
      });
```

Joins

- We can co-ordinate the result of multiple promises using join. here is an example:

```
function doSomething(callBack) {
    setTimeout(function() {
        console.log("2 seconds up");
        callBack(null, "2 seconds up");
    }, 2000);
}
function doSomethingElse(callback) {
    setTimeout(function() {
        console.log("5 seconds up");
        callback(null, "5 seconds up");
    }, 5000);
}

var doSomethingWithPromise = Promise.promisify(doSomething);
var doSomethingElseWithPromise = Promise.promisify(doSomethingElse);

Promise.join(doSomethingWithPromise(), doSomethingElseWithPromise(), function(val1, 
val2){
    console.log("Joined: "+val1);
    console.log("Joined: "+val2);
});
```

Print all files in directory /Users/folder1 and /Users/folder2 together using our command processor

Map

- Map allows an array to be transformed to another array using a function that can transform and individual element using an async promise, using a specific concurrency

```
var fileNames = ["package.json", "package2.json"];
Promise.map(fileNames, function(fileName) {
    return fs.readFileAsync(fileName)
        .then(JSON.parse)
        .caught(SyntaxError, function(e) {
            e.fileName = fileName;
            throw e;
        });
},{concurrency: 1}).then(function(parsedJSONs) {
    console.log(parsedJSONs);
}).caught(SyntaxError, function(e) {
   console.log("Invalid JSON in file " + e.fileName + ": " + e.message);
});
```

Reduce

- Calls a function with each promise return value to arrive at a cumulative aggregate value.

- Reduction function is called as soon as any promise result is ready (unlike join/all)

```
Promise.reduce(
      [ promise1, promise2 ],
      function(reducedResult, result) {
         return reducedResult + result.value
      }, 0).then(function(reducedResult) {
            console.log("Final reduced result: " + reducedResult);
      });
```

Write a new folder size command in CommandProcessor using the reduction function. Use fs.readAsync method

Functional Reactive Programming

- Functional reactive programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter)

Where is this useful

- When there are too many events from diverse sources manipulating too many state fields
- We bundle events into different pipes. And attach functional programs to these pipes to perform actions.
- Events evolve with map, filter and reduce

Event Stream Concept

- First concept of FRP is assemble happenings in the system as a stream of events
  - Events can come from a promise (A single event and end of stream): Bacon.fromPromise(promise)
  - From Node.js event emitters: Bacon.fromEventTarget(eventEmitter,eventName)
  - Single event from a function that takes a callback: Bacon.fromNodeCallback(f)
  - Bacon.fromPoll(interval, f): f should return Bacon.next or Bacon.end. f is called in intervals

Lets Try It

- Attach a Bacon stream to file read stream and read via that stream to find file size using the method Bacon.fromEventTarget(stream, eventName); and event.onValue()

```
function readFile() {
    fs.stat(path, function(err, stat) {
        if (err !== null) {
            callBack(err);
        }
        console.log("Opening read stream...");
        var readStream = fs.createReadStream(path);

        //Replace below section
        readStream.on("data",function(chunk){
                console.log("File stream data event "+chunk.length);
                dataSize+=chunk.length;
        });
    });
}
```

Properties

- Properties are very similar to streams except that they have a current value.
- Properties are result of “reduce” operations on some stream
- property.sample(interval) - get current value at certain intervals
- property.sampledBy(stream) - get current value of property every time there is an event on the stream

Manipulate Streams - Map, Filter, Reduce

- Methods on streams: .onValue, .onError, .onEnd,
- .map(function(value){}) - Converts events in this stream using the provided function
- .map(property) - Puts property into stream for every event on stream
- .filter(function(value){}) - retains or discards events based on return of this function (true/false)
- .filter(property)

Avoid The Scope Var

- Use a property to refer to the data. Using a reduction operation called scan

```
var dataProperty = dataStream.scan(initialValue,function(accumulation,streamData){
       //Create new accumulated value using previous accumulation + content of
streamdata
});
```

- Sample the property only when we have the end event: var reducedStream = dataProperty.sampledBy(endStream);

Custom Streams

```
var stream = Bacon.fromBinder(function(sink) {
     sink("first value")
     sink([new Bacon.Next("2nd"), new Bacon.Next("3rd")])
     sink(new Bacon.Next(function() {
       return "This one will be evaluated lazily"
     }))
     sink(new Bacon.Error("Error value"))
     sink(new Bacon.End())
     return function() {
        // unsub functionality here
     }
   });


var connectionsStream = Bacon.fromBinder(function(sink) {
   io.on('connection', sink);
});
```

Create Additional Streams

- observable.flatMap(f) for each element in the source stream, spawn a new stream using the function f. Collect events from each of the spawned streams into the result EventStream.

Reduce

- .scan(seed, f) scans stream/property with given seed value and accumulator function, resulting to a Property.

```
var plus = function (a,b) { return a + b }
var summedProperty = Bacon.sequentially(1, [1,2,3]).scan(0, plus)
```

- .reduce(seed, f) is like scan but only emits the final value, i.e. the value just before the observable ends. Returns a Property.

- observable.diff(start, f)

Combine Streams & Properties

- observable.combine(property2, f) combines the latest values of the two streams or properties using a two-arg function. The result is a Property.

Case Study Design

- Load balanced command server code review and FRP design

Memory Leaks

- Restarting the application or throwing more RAM at it is all that is needed and memory leaks aren’t fatal in Node?

- As leaks grow, V8 becomes increasingly aggressive about garbage collection

- Closures and Globals are common areas for memory leaks

Some tools

- node-heap-dump takes a snapshot of the V8 heap and serializes the whole thing out in a huge JSON file. It includes tools to traverse and investigate the resulting snapshot in JavaScript.

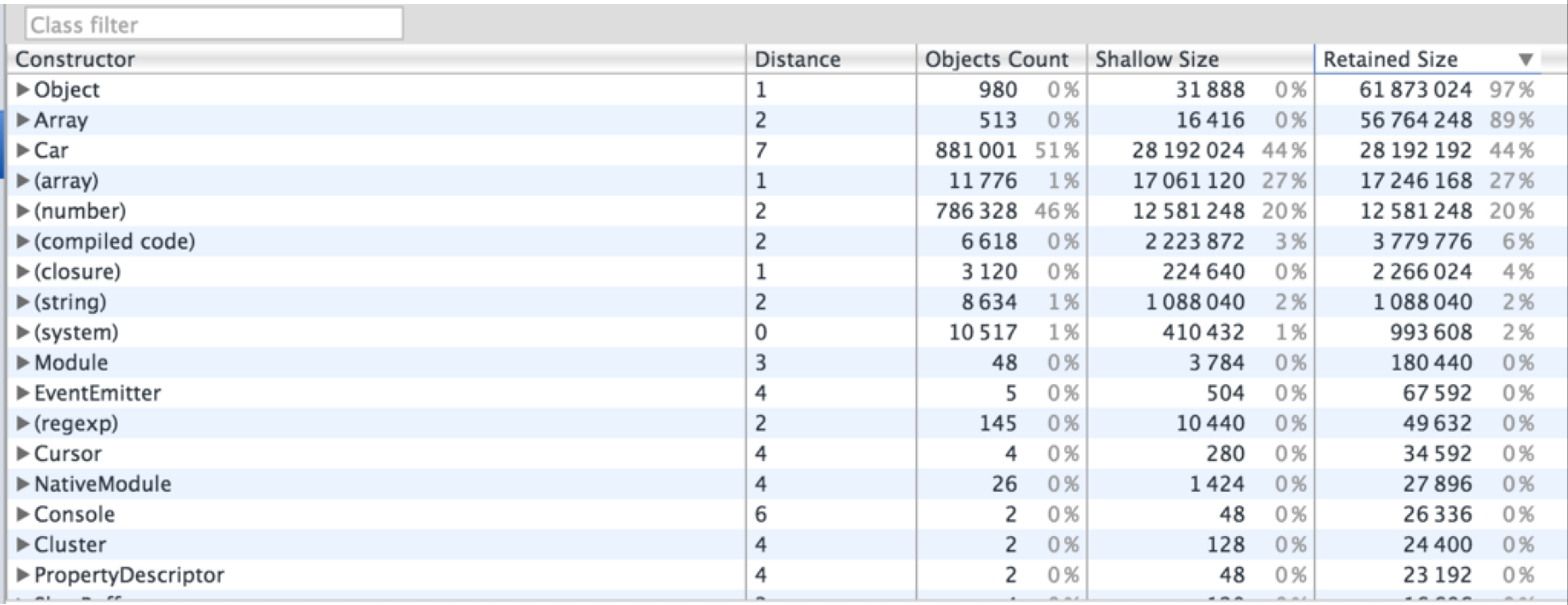
- Node Inspector is a debugger interface for Node.js applications that uses the Blink Developer Tools (formerly WebKit Web Inspector).

- node-memwatch - this is really cool tool for use for live memory watching and analysis

node-inspector

- npm install -g node-inspector
- node-debug yourapp.js
- This launches in your default browser.
  - Can debug
  - Can profile CPU
  - Can analyse heap

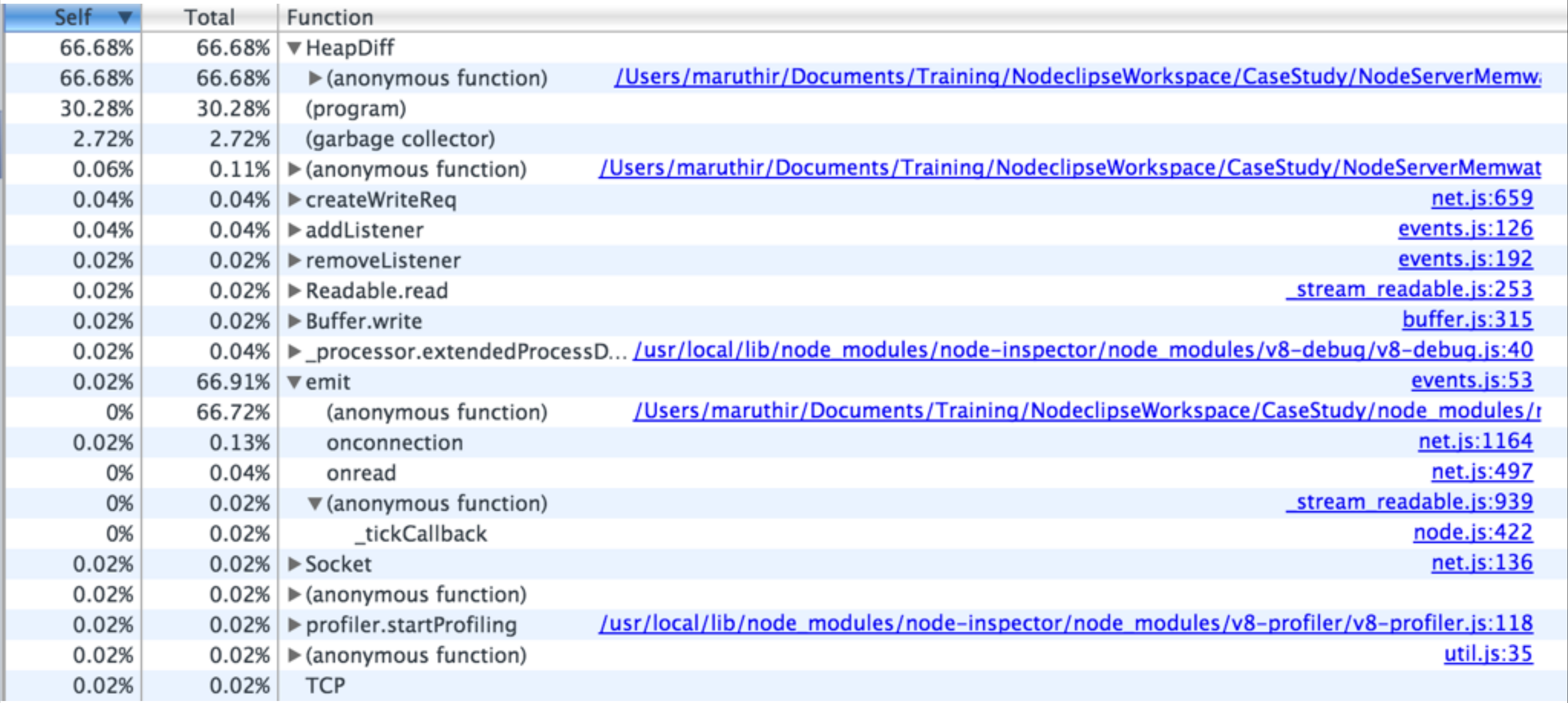
Nodeinspector Mem Profile



Class filter

Constructor	Distance	Objects Count		Shallow Size		Retained Size ▼	
▶ Object	1	980	0 %	31 888	0 %	61 873 024	97 %
▶ Array	2	513	0 %	16 416	0 %	56 764 248	89 %
▶ Car	7	881 001	51 %	28 192 024	44 %	28 192 192	44 %
▶ (array)	1	11 776	1 %	17 061 120	27 %	17 246 168	27 %
▶ (number)	2	786 328	46 %	12 581 248	20 %	12 581 248	20 %
▶ (compiled code)	2	6 618	0 %	2 223 872	3 %	3 779 776	6 %
▶ (closure)	1	3 120	0 %	224 640	0 %	2 266 024	4 %
▶ (string)	2	8 634	1 %	1 088 040	2 %	1 088 040	2 %
▶ (system)	0	10 517	1 %	410 432	1 %	993 608	2 %
▶ Module	3	48	0 %	3 784	0 %	180 440	0 %
▶ EventEmitter	4	5	0 %	504	0 %	67 592	0 %
▶ (regexp)	2	145	0 %	10 440	0 %	49 632	0 %
▶ Cursor	4	4	0 %	280	0 %	34 592	0 %
▶ NativeModule	4	26	0 %	1 424	0 %	27 896	0 %
▶ Console	6	2	0 %	48	0 %	26 336	0 %
▶ Cluster	4	2	0 %	128	0 %	24 400	0 %
▶ PropertyDescriptor	4	2	0 %	48	0 %	23 192	0 %

Nodeinspector CPU profile



Self ▼	Total	Function	
66.68%	66.68%	▼ HeapDiff	
66.68%	66.68%	▶ (anonymous function)	/Users/maruthir/Documents/Training/NodeclipseWorkspace/CaseStudy/NodeServerMemw
30.28%	30.28%	(program)	
2.72%	2.72%	(garbage collector)	
0.06%	0.11%	▶ (anonymous function)	/Users/maruthir/Documents/Training/NodeclipseWorkspace/CaseStudy/NodeServerMemwat
0.04%	0.04%	▶ createWriteReq	net.js:659
0.04%	0.04%	▶ addListener	events.js:126
0.02%	0.02%	▶ removeListener	events.js:192
0.02%	0.02%	▶ Readable.read	_stream_readable.js:253
0.02%	0.02%	▶ Buffer.write	buffer.js:315
0.02%	0.04%	▶ _processor.extendedProcessD…	/usr/local/lib/node_modules/node-inspector/node_modules/v8-debug/v8-debug.js:40
0.02%	66.91%	▼ emit	events.js:53
0%	66.72%	(anonymous function)	/Users/maruthir/Documents/Training/NodeclipseWorkspace/CaseStudy/node_modules/r
0.02%	0.13%	onconnection	net.js:1164
0%	0.04%	onread	net.js:497
0%	0.02%	▼ (anonymous function)	_stream_readable.js:939
0%	0.02%	_tickCallback	node.js:422
0.02%	0.02%	▶ Socket	net.js:136
0.02%	0.02%	▶ (anonymous function)	
0.02%	0.02%	▶ profiler.startProfiling	/usr/local/lib/node_modules/node-inspector/node_modules/v8-profiler/v8-profiler.js:118
0.02%	0.02%	▶ (anonymous function)	util.js:35
0.02%	0.02%	TCP	

node-memwatch

- npm install memwatch

```
memwatch.on('leak', function(info) {});

memwatch.on('stat', function(info) {});

heapDiff = new memwatch.HeapDiff();

var diff = heapDiff.end();
console.log("Heap Diff: %j",diff);
```

memwatch result

```
Memory leaking: {"start":"2015-02-16T03:38:04.000Z","end":"2015-02-16T03:38:09.000Z","growth":84764232,
    "reason":"heap growth over 5 consecutive GCs (5s) - -2147483648 bytes/hr"}

Memory leaking: {"start":"2015-02-16T03:38:18.000Z","end":"2015-02-16T03:39:01.000Z","growth":189013080,
    "reason":"heap growth over 5 consecutive GCs (43s) - -2147483648 bytes/hr"}

Heap Diff: {
    "before":{"nodes":2816350,"time":"2015-02-16T03:38:09.000Z","size_bytes":94757104,"size":"90.37 mb"},
    "after":{"nodes":9674556,"time":"2015-02-16T03:39:29.000Z","size_bytes":314793336,"size":"300.21
mb"},
    "change":{"size_bytes":220036232,"size":"209.84 mb","freed_nodes":4005,"allocated_nodes":
6862211,"details":
        [{"what":"Arguments","size_bytes":0,"size":"0 bytes","+":1,"-":1},
         {"what":"Array","size_bytes":55748280,"size":"53.17 mb","+":1375,"-":1510},
         {"what":"Buffer","size_bytes":2640,"size":"2.58 kb","+":55,"-":0},
         {"what":"Car","size_bytes":109735456,"size":"104.65 mb","+":3429233,"-":0},
         {"what":"Closure","size_bytes":8568,"size":"8.37 kb","+":369,"-":250},
         {"what":"Code","size_bytes":-292992,"size":"-286.13 kb","+":19,"-":528},
         {"what":"Date","size_bytes":0,"size":"0 bytes","+":2,"-":2},
         {"what":"InternalArray","size_bytes":0,"size":"0 bytes","+":1,"-":1},
         {"what":"Native","size_bytes":0,"size":"0 bytes","+":1,"-":1},
         {"what":"Number","size_bytes":54876992,"size":"52.33 mb","+":3429815,"-":3},
         {"what":"Object","size_bytes":3160,"size":"3.09 kb","+":265,"-":134},
         {"what":"ReadableState","size_bytes":9792,"size":"9.56 kb","+":150,"-":99},
         {"what":"SlowBuffer","size_bytes":0,"size":"0 bytes","+":1,"-":1},
         {"what":"Socket","size_bytes":4232,"size":"4.13 kb","+":150,"-":127},
         {"what":"String","size_bytes":-8624,"size":"-8.42 kb","+":5,"-":221},
         {"what":"TCP","size_bytes":1600,"size":"1.56 kb","+":150,"-":100},
         {"what":"WritableState","size_bytes":18720,"size":"18.28 kb","+":150,"-":33}]}}
```

NPM Nuances

- Start projects with npm init to create a package.json
- Install dependencies with a --save or —save-dev option to put it in the package.json: npm install express —save
- Specify a start file so we can start apps with npm start (Which will then automatically create a procfile)

```
"scripts": {
     "start": "node index.js"
   }
```

NPM Nuances

- Specify suitable semver versions for dependencies in package.json such as “~1.9”
- Keep node_modules out of source control
- Use private npm registries if your project is of significant size
- Specify Test Scripts so we can test with npm test

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }
```

NPM Nuances

- Keep track of outdated dependencies with “npm outdated” command

- Use npm scripts for all activities of build and deploy

```
"scripts": {
    "preinstall": "echo Starting app install!",
    "postinstall": "npm run build",
    "build": "grunt"
}
```