Introducing Node.js

Node Training

Node.js Introduction

- How Node.js Works
- Express
- Debugging

Node.js

- Runs high-performance server-side JavaScript
- Uses the Google Chrome V8 engine
  - just-in-time compilation to machine code
  - generation garbage collection (like the Java JVM)
  - creates virtual “classes” to optimize property lookups

Node.js

- Provides a minimal system level API for networking, file system, event handling, streaming data, and HTTP/S.
- Has a well-designed module system for third party code - very effective and simple to use
- Code runs in a single non-blocking JavaScript thread
  - Most of the time we're waiting for the database or network anyway!

A Simple Node.js Web Server

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

var server = http.createServer(function(req, res) {

 res.writeHead(200, {'Content-Type': 'text/plain'});
 res.end('Hello World\n');

});

server.listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337');
```

```
$ node server0.js
```

```
$ ab -c 100 -n 10000 http://127.0.0.1:1337/
```

Up and Running

```
$ node server0.js
```

- Download Node.js from http://nodejs.org
- Run the example server
- Open the web page
- http://127.0.0.1:1337
- Does it work?

V8 JavaScript Engine

- Embeddable C++ Component
  - multi-platform: Windows, Linux, Mac
- Standards Compliant
  - ECMAScript-262, 5th Edition (use --harmony for 6th Ed.)
- Does not provide DOM API
  - The embedding environment (i.e. browser) should provide this, or not (Node.js)

V8 JavaScript Engine

- Can expose C++ objects to JavaScript
  - Node.js uses this extensively
    - e.g. HTTP Parser
- Designed for speed
- The Chrome Comic Book:
  - http://www.google.com/googlebooks/chrome/big_12.html
  - read pages 12-17

Dynamic Code Generation

- Many languages create an intermediate representation
  - byte code (e.g. Java)
  - or internal memory structures
    - typically representing the Abstract Syntax Tree
- Code execution requires interpretation of these structures is slow

Dynamic Code Generation

- V8 compiles JavaScript directly to machine code
  - dynamically patching as needed
    - when inline caching fails
    - when profiling indicates a "hot spot"

Generational Garbage Collection

- Automatic Memory Management
  - "old" objects that no other objects depend on are removed from memory
- V8 Garbage Collector stops execution at regular intervals to check a subset of objects

Generational Garbage Collection

- Uses generations
  - young: short lived objects
    - need to be checked frequently
  - old: longer term objects
    - objects end up here if they survive the young generation
    - need to be checked less frequently

Virtual Classes

- Traditionally, objects are hash maps
  - continuous lookups are slow
  - traversing prototype chains is slow
- V8 creates a hidden class:
  - Each time a property is added to an object
    - a new hidden class is created
    - or, if found, a matching hidden class is used

Virtual Classes

- With classes, property lookups do not require a hash table
- Hidden classes allow for class-based optimizations:
  - inline-caching: store previously resolved methods

Events and Callbacks

- Node.js runs an Event Loop to handle Input/Output:
  - While there's another event pass it to a callback function
- OS notifications are the basis of IO events
  - each time some data arrives from outside, trigger an event
  - partial data can arrive from many different clients in any sequence - lower level modules handle buffering - libuv

Events and Callbacks

- Callbacks should handle data as quickly as possible and return
- This is usually what happens in web servers:
  - load data from database, format as JSON, send it out
- Single thread for application code
  - Your JavaScript code runs by itself
  - offload CPU intensive tasks or you'll cause delays
- Internally, C++ threads achieve this

Blocking Code

```
Statement stmt = conn
  .createStatement();

ResultSet rs = stmt
.executeQuery("SELECT * FROM customers");
```

- Traditional code waits for input before proceeding
- The Java thread above "blocks" on `executeQuery`

Blocking Code

```
Statement stmt = conn
  .createStatement();

ResultSet rs = stmt
.executeQuery("SELECT * FROM customers");
```

- Execution doesn't proceed until the database returns a result
- The thread consumes large amounts of resources:
  - memory for the stack
  - CPU for context switching

Non-blocking Code

```
collection.findOne(query, function( err, result ) {
    ...
});
```

- Callback-based code waits for events
- Executing the callback is cheap
  - Just a JavaScript function call with a given context

Non-blocking Code

```
collection.findOne(query, function( err, result ) {
    ...
});
```

- There is no thread management
  - Just execute callbacks when there are events
- Callback functions are easier to use when the language supports anonymous functions

The express Module

```
$ npm init
$ npm install express --save
$ node express0.js
```

- General purpose web server
  - handles most of the things you need for HTTP
  - simple plugin pattern: “middleware"
    - shares this pattern with the `connect` module
- Open http://localhost:3000

URL Routing

```
$ node express1.js
```

- Express lets you
  - route on URL patterns
  - grab parameters from the URL
  - Open http://localhost:3000/say/alice

Static Files

```
var app = express();
app.use(express.static(__dirname + '/public'));
```

- You can serve static files with the built-in static “middleware”
- Middleware order is significant, the static middleware is usually included last to allow dynamic routes to take precedence
- Run below and open http://localhost:3000

```
$ node express2.js
```

JSON API

```
$ node express3.js
```

- The `body-parser` middleware lets you work with JSON in a simple manner
  - Open http://localhost:3000/ping
  - Use curl as below to send a `POST` request

```
$ curl -X POST -H "Content-Type: application/json"
  -d '{"foo":1}' http://localhost:3000/print
```

With Callbacks

```
$ node express4.js
```

- Load a file from disk and return its contents
  - Asynchronously, of course
  - This is the most common pattern
- Open http://localhost:3000/load

Custom Middleware

```
app.use(function(req, res, next) { ... })
```

- Writing middleware is trivial
  - Just give express a function with signature:
  - Call next if the middleware isn't sending a response
  - Open http://localhost:3000/slow

```
$ node express5.js
```

Debugging Node

```
debugger;
```

- Node comes with a built in command-line debugger
  - You'll need to know how to use it
  - It might be your only option on a production server

```
$ node debug debug0.js
```

Node Inspector

```
$ npm install -g node-inspector
$ node-debug debug0.js
```

- Node-inspector provides a Chrome Devtools interface
  - http://github.com/node-inspector/node-inspector

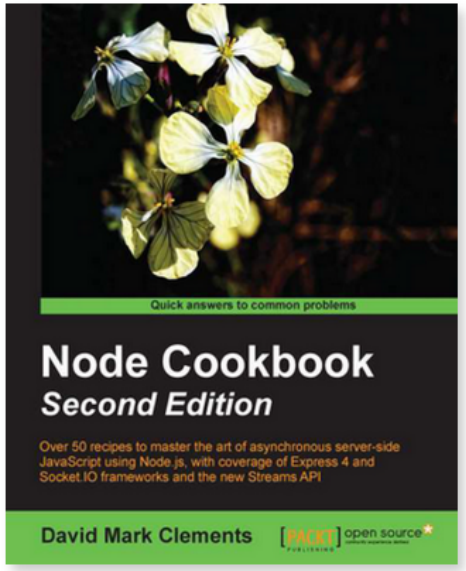
Profiling

```
$ npm i -g tick
$ node --prof profile0.js
$ node-tick-processor
```

- V8 profile output requires tool analysis
- NodeTime provides a graphical interface
  - http://nodetime.com Commercial service with free option
  - Run uglier, less capable solution locally without signup to Nodetime using the `look` module
  - https://thlorenz.github.io/v8-perf/ has a wealth of info profiling and V8 optimisations info

Where Next?

- Try the basic lessons:
  - http://nodeschool.io/#learn-you-node
  - http://nodeschool.io/#expressworks
- Node Cookbook ;)
  - http://amazon.co.uk/dp/1783280433



nearForm