latex input: mmd-article-header Title: NodeJS Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Node.js, Express.js, Web, Web Framework CSS: http://fletcherpenney.net/css/document.css xhtml header: Copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

Many of these notes come from *Web Development with Node and Express*, by Ethan Brown.

NodeJS

What is NodeJS?

1. A *web server* (like Apache)
2. *Minimal*, meaning easy to set up and configure
3. Single threaded -- you can always spin up more instances of Node
4. Uses Google's V8 JavaScript Engine to JIT compile JavaScript to native machine code
    - Automatically compiled for you behind the scenes
5. Has interfaces to all major relational and NoSQL databases
6. Biggest difference from Apache: the app you write *is* the web server

How does it relate to Nginx fit in here?

- For one thing, on larger projects, Nginx might be your *proxy server* routing requests to a cache or different app instances.
- Other than that, I'm not sure

Intro NodeJS

1. `helloWorld.js`

    ```
    var http = require('http');
    http.createServer( function (req, res) {
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('Hello world!');
    }).listen(3000);
    console.log('Server started on localhost:3000; press Ctrl-C to
    ```

    Now type `node helloWorld.js` and go to `http://localhost:3000`.

    - `req` refers to the *request*, and `res` to the *response*.

2. Handy global varible `__dirname` resolves to the directory the executing script resides in
3. Requiring files
    - To look among `npm` packages (`global` or in `'./node_modules'`)

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

    - To look in current directory

      ```
      var fortune = require('./lib/fortune.js')
      ```

4. `module.exports` is the object that's actually returned as the result of a `require` call. (StOve)
    - `x.js`:

      ```
      module.exports = { a: "hello" };
      ```

    - `y.js`:

      ```
      var x = require('./x');
      console.log(x.a);
      ```

Request Object

1. **Parameters** can come from
    1. querystring
    2. session (cookies)
    3. request body (POST)
    4. named routing parameters (`'/:name'`)
2. Book author says
    1. normal `req.param` method munges all parameters together
    2. so avoid it
    3. instead, use Express's explicit parameter-holding properties

Hooking in MongoDB

1. "While there's a low-level driver available for MongoDB , you'll probably want to use an **'object document mapper' (ODM)**. The officially supported ODM for MongoDB is **`Mongoose`**."

ExpressJS

What is ExpressJS?

1. A web framework (a bit) like Ruby on Rails
    - Not the *only* web framework for Node, but pretty dominant right now
2. Written by the one and only TJ Holowaychuk
3. Default **templating engine** is (TJ's) `Jade`, which is dope
4. It has **scaffolding** (boilerplate generation via script), inspired by Rails

Intro ExpressJS

1. Here is an example route

    ```
    app.get('/ab', function(req, res) {
      res.type('text/plain');
      res.send('Meadowlark Travel');
    });
    ```

    This says,

    1. Route any of the following HTTP `GET` paths to the callback
        - `/ab`
        - `/ab/`
        - `/ab?cd=ef`
        - `/ab/?cd=ef`
    2. Set the `Content-Type` header to `'text/plain'`
    3. end the `response` by putting the given text through the wire over TCP
2. `app.use` adds "*middleware*" to Express
3. Note, routes and middleware are added **in order**
4. To render a template and pass in a variable, we have

    ```
    router.get('/', function(req, res) {
      res.render('index', { title: 'Express' });
    });
    ```

5. The `static` middleware makes serving static files easier
6. By default, Express looks for *views* to render (e.g. `index` above) in the `views/` directory
    - It looks for *layouts* (html reused on multiple pages) in `views/layouts/`

Request/Response object

1. Requests start as an instance of Node's `http.IncomingMessage`, to which methods are added
2. Responses start as instances of Node's `http.ServerResponse` objects.
3. `res.send(body)`, `res.send(status, body)`
    1. defaults to a content type of `text/html` so if you want to change it, call `res.set('Content-Type', 'text/plain')` or `res.type('txt')` before `res.send`
    2. If body is an `object` or `array`, the response is sent as `JSON`
        - Though you *should* ***explicitly*** send `JSON` using `res.json(json)`
4. `res.query` -- querystring values
5. `req.session` -- session values
6. `req.cookie/req.signedCookies` -- cookies
7. `res.render` -- render a view within a layout

Middleware

1. *Middleware is a function* that takes three arguments:
    1. Request object
    2. Response object
    3. "Next" function
2. Executed in a *pipeline* -- order matters: things added by one middleware are available to everyone downstream
3. Insert middleware into the pipeline with `app.use`
4. **Don't forget to call `next()`** -- otherwise the request will terminate!

```
app.use(function(req, res, next) {
  console.log('processing request for "' + req.url + '"....');
  next(); // <-= NB
});
```

5. Add middleware to specific verbs with `app.VERB`

```
app.get('/b', function(req, res, next) {
  console.log('/b: route not terminated');
  next();
});
```

Intermediate level

Useful packages

1. fs -- this is a "core" node module (ie. it's part of the std-lib)
2. path
3. temp -- useful for temp files and dirs e.g. as part of `grunt` tasks
4. express
5. http -- one of the "core" modules, let's you listen on ports, shove data through sockets, add headers, etc.
6. child_process -- spawn child processes to execute given commands
7. async
8. underscore -- array combinators
9. mkdirp -- exactly whatcha think
10. optimist -- options parsing
11. rimraf -- `rm -rf` for node

Testing

1. jasmine-node

Jasmine Testing

Testing asynchronous stuff

Use `waitsFor(condName, timeMS, function(){booleanConditions;})` and `runs(func(){expects;})`

If you have a `runs` after a `waitsFor`, the runs won't execute until the `waitsFor` conditions have either been met or have timed-out.

Expecting values

`expect(something).toBeTruthy()`

Why don't we just `expect(something).toBe(true)`?

Because *many* things are `truthy`, but only `true === true`. Things that are `truthy` include anything that is *not* one of

- false
- 0
- ""
- undefined
- null
- NaN

Sure, this is a bit gross but it also makes some sense.

Spies

Let you determine what methods have/not been called on a given object.

```
spyOn(console, 'log')
spyOn(console, 'error')

# Now you can
expect(console.error).toHaveBeenCalled()
expect(console.log).not.toHaveBeenCalled()

# the first argument of the first time `error` was called
# should "match" argVal.length > 0
expect(console.error.argsForCall[0][0].length).toBeGreaterThan 0
```

Express

Here's an *entire* minimal REST endpoint for a file server. It may be a bit out of date in the API it uses (viz. I think `sendfile` is now `sendFile`), but you get the gist.

```
app = express()
app.get '/node/v0.10.3/node-v0.10.3.tar.gz', (request, response) ->
  response.sendfile path.join(__dirname, 'fixtures', 'node-v0.10.3.t
app.get '/test-module-1.0.0.tgz', (request, response) ->
  response.sendfile path.join(__dirname, 'fixtures', 'test-module-1
server =  http.createServer(app)
server.listen(3000)
```

This server can be shut-down programmatically (e.g. for a Jasmine test) using `server.close()`.

Grunt

It's a task executor, like make, rake, sbt etc. You define what the tasks can do and then you can execute them with `grunt taskName`. You can also load tasks directly from npm using `grunt.loadNpmTasks('name-of-task')`.

Using CoffeeScript

In the base directory, in your `Gruntfile.coffee`, have the following (stolen from the apm -- Atom Package Manager)

```
module.exports = (grunt) ->
  grunt.initConfig
    pkg: grunt.file.readJSON('package.json')

    coffee:
      glob_to_multiple:
        expand: true
        cwd: 'src'
        src: ['*.coffee']
        dest: 'lib'
        ext: '.js'

  grunt.loadNpmTasks('grunt-contrib-coffee')

  grunt.registerTask 'clean', -> require('rimraf').sync('lib')
  grunt.registerTask('default', ['coffee'])
```

Explained

We're loading the npm package `grunt-contrib-coffee`, which is going to look at the preferences specified within the `coffee` object to configure where it will look for the `.coffee` files, and where it will put the compiled `.js` files. The package registers itself as a task called `'coffee'`

Then we register `'default'` to run `'coffee'`, meaning we just have to run `grunt` at the command line to recompile all the `coffeescript` files.

modules

the docs

module.exports

Here we're assuming you followed the coffeescript directions above.

When you want it to be possible to do one of these

```
# this is like "import my_lib" in python
usefulFunctions = require('../lib/functionDefs')
myWorkIsDone = usefulFunctions.wowSoGladIGotThis()

# or you could emulate "from a import b" like this
greatFunc = require('../lib/functionDefs').wowSoGladIGotThis
```

Then in `src/functionDefs.coffee`, you must attach the function objects that you want to *export* to the `module.exports` object.

```
greatFunc = -> myFavoriteThings()
module.exports.wowSoGladIGotThis = greatFunc
```

using directories

If your library is in a self-contained directory, it needs an entry-point so that it can be `require`'d. Say you want to be able to `require(./library-dir-name)`. One way to allow this is to have a `package.json` file with the following content

```
{ "name" : "my-great-lib", "main" : "my-entrypoint.js" }
```

If there is no `package.json` file, then it will try to load an `index.js` file.

package.json

1. **name** -- whatcha think
2. **description** -- like a little github repo description
3. **version** -- e.g. "0.0.0"
4. **private** -- set to true if you don't want `npm` to make your package widely available
5. **license** -- just use `"license" : "MIT"`
6. **bin** -- use this to link a name from `/usr/local/bin/my-name` to running node `my-supplied-path`
7. **scripts** -- commands to run at various pre-defined times
    1. **prepublish** -- runs before the package is published, and on a local `npm install` without arguments
    2. **test** -- run via the `npm test` command
8. **dependencies** -- required to run
9. **devDependencies** -- required only to devolop
    - e.g. unit tests, coffeescript transpilation, minification, etc.
10. **main** -- fill this in if the object you want imported by `requre('my-package')` is *not* in a file called `index.js`

Dependency versioning syntax

- `~1.2.3` will match all `1.2.x` versions but will miss `1.3.0`
- `^1.2.3` will match all `1.x.x` versions but will hold off on `2.0.0`
- I think you're allowed to literally just say `1.2.x` instead, which is far easier to understand than remembering the crap above
- ref

dependencies vs devDependencies

If you `git clone` a package's source code, and do a `npm install` it assumes you're a developer of that package, and will also install the `devDependencies`. If you `npm install "$package"` it means you just want to *use* the package, and it will only install the `dependencies`.

Node addons & node-gyp

"Addons are dynamically linked shared objects. They can **provide glue to C and C++ libraries**. The API (at the moment) is rather complex, involving knowledge of several libraries."

Basically, if you want your node program to run some C++ you've written, you make an addon. Then you use `node-gyp`, the cross-platform command-line tool written in Node.js to compile the native addon modules for Node.js.

npm

What is npm

1. Node's amazing ubiquitous package manager
2. Stands for "npm is not an acronym" (?? doofii)

How To

1. Install a package *globally* (make it available to your whole system)

    ```
    npm install -g grunt-cli
    ```

2. Save the package(s) in `node_modules/` *and* update the `package.json` file

    ```
    npm install --save express
    ```

3. Save the package in `devDependencies` instead of `dependencies` to reduce dependencies required to deploy (e.g. for *testing*-related modules)

    ```
    npm install --save-dev mocha
    ```