Learn JavaScript & Node.js by exploring npm

by SETH VINCENT

npm recipes

Learn JavaScript and Node.js by exploring npm

Seth Vincent

This book is for sale at http://leanpub.com/npm-recipes

This version was published on 2014-09-07

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

Thank you!

I'm excited to share these tutorials with you! The goal of this series of tutorials is to showcase awesome modules found on npm[1].

There are over 50,000 modules on npm, so there's a wildly big number of opportunities for checking out how various npm modules can work together.

I encourage you to help guide the direction of this book! If there are particular modules that you'd like to see covered, please email me at hi@learnjs.io[2] or create an issue on the project issue queue on GitHub: github.com/learn-js/npm-recipes/issues[3].

This book is highly inspired by the lean publishing model (read more about it[4]) proposed by Peter Armstrong, founder of leanpub.com. It has proven successful as a way to receive feedback from you, the readers, so that together we can make the best book about javascript tools and libraries possible.

Tools we'll use in this book

Many of the tools used in this book are covered in detail in one of my other books, Development Environments for Beginners.

If you're feeling like you could use more information about what a development environment is and how best to set one up, you can purchase the Development Environments book at learnjs.io/books/dev-envs[5].

If needed you can check out the Development Environments book on GitHub for free here: github.com/sethvincent/dev-envs-book[6].

Sublime text editor

Sublime is a popular text editor with versions for Mac, Windows, and Linux.

You can download an evaluation copy for free, and pay for a license when you're ready.

In this book we'll be using version 2 of Sublime, in future updates to the book we'll switch to version 3.

[1] http://npmjs.org
[2] mailto:hi@learnjs.io
[3] http://github.com/learn-js/npm-recipes/issues
[4] https://leanpub.com/manifesto
[5] http://learnjs.io/books/dev-envs
[6] http://github.com/sethvincent/dev-envs-book

Install

Go to sublimetext.com[7] and click the download button.

Terminal

You'll be using the terminal (or command-line) for many of the tutorials in this book. The Terminal chapter in Development Environments for Beginners[8] will help you get started.

Node.js

We will write Node.js-style modules not only for the server, but also the browser.

Our code written for the browser will utilize the node.js style of modules thanks to browserify, a tool for bundling node modules for the browser.

This means that we won't cover the RequireJS/AMD toolset for javascript development, but will focus on Node/CommonJS modules.

npm

npm is the package manager that comes bundled with node.js. This book is all about exploring the modules available on npm. Sometimes people call it the node package manager, but I prefer a term that's becoming more popular: "node packaged modules". The idea is that we can store whatever modules we want on npm. It might be for the server or the client.

In each chapter we'll explore npm a little more, and by the end of the book you'll have learned it pretty thoroughly.

git

I encourage you to use git as version control software to track changes as you complete tutorials. If you're new to git, learn more about it in this Git chapter of the Development Environments book[9], the Try Git interactive tutorial[10], and the Pro Git book that's available for free on the git website[11].

[7] http://www.sublimetext.com/

[8] https://github.com/sethvincent/dev-envs-book/blob/master/manuscript/terminal.md

[9] http://git-scm.com/book

[10] http://try.github.io/

[11] http://git-scm.com/book

GitHub & GitHub Pages

I recommend that you put each project you make while reading this book up on GitHub, and if applicable, GitHub Pages. It'll be good practice if you're new to git and GitHub.

Learn more about GitHub pages in the related chapter of the Development Environments book[12], and on GitHub's Help section[13].

Other books

There's a good chance that if you like this book you'll be interested in the other books in the Learn.js series.

Check them out!

- Introduction to JavaScript & Node.js[14]
- Making 2d Games with Node.js & Browserify[15]
- Mapping with Leaflet.js[16]
- Development Environments for Beginners[17]
- Theming with Ghost[18]

Learn more at learnjs.io[19].

[12] https://github.com/sethvincent/dev-envs-book/blob/master/manuscript/github.md
[13] https://help.github.com/categories/20/articles
[14] http://learnjs.io/books/learnjs-01
[15] http://learnjs.io/books/learnjs-02
[16] http://learnjs.io/books/learnjs-03
[17] http://learnjs.io/books/dev-envs
[18] http://themingwithghost.com
[19] http://learnjs.io

1. Creating standalone JavaScript library builds with browserify, watchify, and uglify-js

Recently I had the opportunity to use Browserify as one of the tools for creating a JavaScript module for a client that is building a mapping product for architects and urban planners.

In that project I used Browserify and npm scripts to bundle the module into a file the client could use as a standalone library that could be added to any web page that needed to use the tool.

It was a fairly straightforward and flexible build process, and here I'll outline a similar structure that you could use in your projects.

Our example project will be named Pizza, because our example library will do nothing but return the string `'Pizza'`. Deal with it.

This post is part of our **npm recipes series**[20], and we'll be looking at the **browserify**[21], **watchify**[22], and **uglify-js**[23] modules. We'll be exploring how to use these three development tools together to make a simple build process.

Create your package.json file

To create a package.json file run this command:

```
1  npm init
```

Answer the questions that it asks (hit enter to keep the default answers) and you should get a package.json file that looks something like this:

[20] http://learnjs.io/npm-recipes
[21] https://github.com/substack/node-browserify
[22] https://github.com/substack/watchify
[23] https://github.com/mishoo/UglifyJS2

```
{
  "name": "library-builds",
  "version": "0.0.0",
  "description": "creating standalone library builds with browserify, watchify, \
and uglify-js",
  "main": "index.js",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "BSD-2-Clause"
}
```

Set up the project files & directories

Create an index.js file.

For this example I'm just making it a function that returns the string `'Pizza'`. Because pizza.

```
module.exports = function(){
  return 'Pizza';
}
```

Create an index.html file

We'll use an overly simple html file for testing out the usage of the library:

```
<!doctype html>
<html>
<head>
  <title>browserify module</title>
</head>
<body>
  <script src="dist/pizza.js"></script>
  <script>
    console.log(pizza())
  </script>
</body>
</html>
```

Create a dist folder

In the dist folder we'll store the pizza.js and pizza.min.js files. You can create it using the `mkdir` command:

```
mkdir dist
```

Install development dependencies

```
npm install --save-dev browserify watchify uglify-js
```

By including the `--save-dev` option these modules will be saved to the `devDependencies` field in your package.json file.

Module usage examples

Using browserify

You can install browserify globally so you can run it's packaged `browserify` command in the terminal:

```
npm install -g browserify
```

Here's a simple example of using the `browserify` command:

```
browserify index.js -o dist/pizza.js
```

The key part of bundling standalone modules with Browserify is the `--s` option. It exposes whatever you export from your module using node's `module.exports` as a global variable. The file can then be included in a `<script>` tag.

You only need to do this if for some reason you *need* that global variable to be exposed. In my case the client needed a standalone module that could be included in web pages without them needing to worry about this Browserify business.

Here's an example where we use the `--s` option with an argument of pizza:

```
browserify index.js --s pizza > dist/pizza.js
```

This will expose our module as a global variable named `pizza`.

To get source maps to make debugging easier, use the `-d` option:

```
browserify index.js -d --s pizza > dist/pizza.js
```

See more options and useful help by running `browserify --help`.

Using watchify

To watch js files for revions you can use watchify, which uses browserify to re-bundle your modules each time you edit a file.

You can install watchify globally so you can run it's packaged `watchify` command in the terminal:

```
npm install -g watchify
```

Basic usage is similar to browserify. Here's what we'll use in this case:

```
watchify index.js -d --s pizza -o dist/pizza.js -v
```

`watchify` uses all the same arguments as `browserify`, with the difference that the `-o` option for the output file is mandatory.

Using uglify-js

We'll probably want to build a minified version of the library, so we can use the uglify-js module for that.

You can install uglify-js globally so you can run it's packaged `uglifyjs` command in the terminal:

```
npm install -g uglify-js
```

Basic usage looks like this:

```
uglifyjs index.js -c > dist/pizza.min.js
```

Note that when using this module on the command line there's no dash. Just `uglifyjs`. That's messed me up before.

The `-c` option compresses the output.

To see all options and get some nice help text, you can run `uglifyjs --help`.

To use the `uglifyjs` command with the `browserify` command, we'll pipe the output of browserify to uglify-js:

```
browserify index.js --s pizza | uglifyjs -c > dist/pizza.min.js
```

Let's take a look at how we can move these commands into the npm scripts

Add build scripts to the "scripts" field

Add the example usages we created above to the `scripts` field of your package.json file so that the `scripts` field looks like this:

```
"scripts": {
  "build-debug": "browserify index.js -d --s pizza > dist/pizza.js",
  "build-min": "browserify index.js --s pizza | uglifyjs -c > dist/pizza.min.js",
  "build": "npm run build-debug && npm run build-min",
  "watch": "watchify index.js -d --s pizza -o dist/pizza.js -v"
},
```

Run the build scripts!

Create the debug build of the library:

```
npm run build-debug
```

Create the minified build:

```
npm run build-min
```

Create both the debug and minified builds:

```
npm run build
```

Watch the main file for changes and automatically regenerate the debug build:

```
npm run watch
```

Usage of the library

All we have to do is include the dist/pizza.js or dist/pizza.min.js files in the page, then use that `pizza` variable that was exposed using Browserify.

In our case, pizza is a function that returns the string `Pizza`. For you, it might be a constructor function, object, or whatever you need to provide the necessary functionality.

More like this

Learn more about using npm scripts by taking a look at this article by the author of browserify: task automation with npm run[24].

For an example of running similar tasks for bundling css that's packaged through npm, check out the next chapter: "Using rework-npm for bundling css from npm along with myth and clean-css"!

[24] http://substack.net/task_automation_with_npm_run

Using rework-npm for bundling css from npm along with myth and clean-css

In this tutorial we'll focus on bundling css that's published on npm using three modules:

- rework-npm-cli[25], a module for `@importing` css from the `node_modules` folder, based on rework-npm[26].
- myth[27], a preprocessor module that generates cross-browser css based on rework[28].
- clean-css[29], a module that minifies css files.

To get started, open a terminal to make a directory for your project and move into it:

```
mkdir my-project-name
cd my-project-name
```

Now create a package.json file using `npm init`

```
npm init
```

This command will ask you questions about your project. Answer all the questions and it'll create a package.json file for you.

Install dependencies

Install rework-npm-cli, myth, and clean-css using the `npm install` command:

```
npm install --save-dev rework-npm-cli myth clean-css
```

The `--save-dev` option saves these modules as development dependencies to your package.json file.

[25] https://github.com/sethvincent/rework-npm-cli
[26] https://github.com/conradz/rework-npm
[27] https://github.com/segmentio/myth
[28] https://github.com/reworkcss/rework
[29] https://github.com/GoalSmashers/clean-css

Create project files.

We'll need two files for now: source.css and index.html.

Create them using the terminal:

```
touch source.css index.html
```

The `touch` command creates a file if it doesn't already exist.

Create the build script in package.json

In your package.json file you should have a scripts field that looks something like this:

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

Revise it to look like this:

```
"scripts": {
    "bundle": ""
  },
```

Next we'll look into the functionality of rework-npm-cli, myth, and clean-css, and create the build script using commands that they expose.

Using rework-npm-cli

The rework-npm-cli module exposes a `rework-npm` command that allows usage of the rework-npm[30] module on the command line, and typical usage looks like this:

```
rework-npm source.css -o bundle.css
```

Add the rework-npm-cli command to the start script in the package.json file so it looks like this:

[30] https://github.com/conradz/rework-npm

```
"scripts": {
  "bundle": "rework-npm source.css -o bundle.css"
},
```

Now we can run `npm run bundle` to generate a bundle.css file.

To test out the functionality of the `rework-npm` command, let's install a sample package I created called skelestyle-typography[31].

```
npm install --save skelestyle-typography
```

Now add an import statement to your source.css file:

```
@import "skelestyle-typography";
```

Note that rework-npm requires double quotes.

Now bundle the css

```
npm run bundle
```

You'll see the contents of the skelestyle-typography css package in your bundle.css. It should look something like this:

```
body,
button,
html,
input,
select,
textarea {
  font-size: 20px;
  line-height: 1.6;
  font-family: Georgia,serif;
  font-smoothing: antialiased;
  font-weight: 300;
  color: #444;
}

::-moz-selection {
  color: #323230;
```

[31] https://github.com/sethvincent/skelestyle-typography

```
  background-color: #d6d6d6;
  text-shadow: none;
}

::selection {
  color: #323230;
  background-color: #d6d6d6;
  text-shadow: none;
}

*,
:after,
:before {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

h1,
h2,
h3,
h4,
h5,
h6 {
  font-family: 'Helvetica Neue',Helvetica,sans-serif;
  text-rendering: optimizeLegibility;
  line-height: 1.2;
  margin-top: 0;
  margin-bottom: 5px;
}

.post h1,
.post h2,
.post h3,
.post h4,
.post h5,
.post h6 {
  margin-top: 40px;
}

a {
  color: #3c80c3;
```

```
59    -webkit-transition: color ease .2s;
60    transition: color ease .2s;
61    text-decoration: none;
62  }
63
64  a:hover {
65    color: #57A3E8;
66  }
67
68  a:active,
69  a:focus {
70    color: #1e4062;
71  }
72
73  p {
74    margin-top: 5px;
75    margin-bottom: 10px;
76  }
77
78  code {
79    font-family: Monaco,Menlo,monospace;
80    padding: 5px 10px;
81    background-color: #fdfdfb;
82    border: 1px solid #dadad8;
83    font-size: 13px;
84    border-radius: 2px;
85    white-space: nowrap;
86  }
87
88  pre {
89    font-family: Monaco,Menlo,monospace;
90    font-size: 13px;
91    padding: 10px 15px;
92    background-color: #fdfdfb;
93    border: 1px solid #dadad8;
94    border-radius: 3px;
95    overflow-x: auto;
96    margin-bottom: 35px;
97  }
98
99  pre code {
100   background-color: none;
```

```
  border: 0;
  padding: 0;
  white-space: pre;
}
```

Create an index.html file

Create an index.html file and add this content:

```
<html>
<head>
  <meta charset="utf-8">
  <title>rework-npm, myth, & clean-css.</title>
  <link rel="stylesheet" href="bundle.css">
</head>
<body>

<h1>Site title</h1>
<h3>Subheader</h3>

<p>A bunch of text that is here to see what text looks like.</p>

</body>
</html>
```

Check out this site in a browser and you'll see the skelestyle-typography styles have been included and applied.

Overriding values

You can override any css rule just like normal. For example:

```
@import "skelestyle-typography";

body {
  color: red;
}
```

Add the above body statement to your source.css file, now bundle the css again:

```
npm run bundle
```

You'll see that the body statement you added is at the bottom of bundle.css, and if you check out index.html in the browser again, you're added css is making all the text red.

Let's extend this example now with the myth command line tool.

Using myth

myth[32] is described on github as a "A CSS preprocessor that acts like a polyfill for future versions of the spec."

So we can use upcoming css features now!

As an example, let's add some variables to source.css:

```
@import "skelestyle-typography";

:root {
  var-gray: #323232;
  var-serif: Georgia, serif;
}

body {
  color: var(gray);
  font-family: var(serif);
}
```

Bundle the css again with `npm run bundle`

Now you'll see this at the bottom of your bundle.css file:

```
body {
  color: #323232;
  font-family: Georgia, serif;
}
```

It worked!

Read more about the funtionality provided by myth[33]. I've found that it provides all I need from a CSS preprocessor.

Now let's add in another command line tool to minify the css.

[32] https://github.com/segmentio/myth

[33] https://github.com/segmentio/myth

Using clean-css

clean-css[34] is great for minifying css files, and fits in well with the other tools we've looked at so far.

First, revise the bundle script in the package.json file to pipe the output of `myth` to the `cleancss` command:

```
"scripts": {
  "bundle": "rework-npm source.css | myth | cleancss -o bundle.css"
},
```

Now, run `npm run bundle` again, and your bundle.css file is now minified!

You've probably grown tired of running `npm run bundle` over and over amiright? Let's take a look at how that can be automated.

Watching files with nodemon

We can use the nodemon command line tool to watch files for changes and automatically run the bundle script.

First, install nodemon:

```
npm install --save-dev nodemon
```

Next, add a `start` script to the scripts field in your package.json file:

```
"start": "nodemon -e css --ignore bundle.css --exec 'npm run bundle'"
```

This watches all files with an extension of css, and executes the `npm run bundle` command each time a css file is changed. We ignore bundle.css because otherwise we'd likely get stuck in a loop of updating files.

Run `npm start` and edit the source.css file. Every time you save you'll see the bundle.css file update with your new changes!

[34] https://github.com/GoalSmashers/clean-css

Create a Node.js web app that pulls data from content APIs using hyperquest, director, handlebars, and other modules from npm

The goal for this tutorial is to create a server-side Node.js application that pulls data from a content API and serves it to the browser as HTML. We'll use the API of a LocalWiki[35] site, SeattleWiki.net[36], as the content API.

First we'll run through example usage of each of the modules used in this project, then we'll build the actual application.

Our project will use these modules:

- hyperquest[37] - for requesting data from the API
- director[38] - for routing requests to the server
- handlebars[39] - for the HTML templates served to the browser
- handlebars-layouts[40] - so we can have jade/django style layouts using handlebars
- st[41] - for serving static files
- event-stream[42] - for working with the data stream we get back from hyperquest
- rework-npm-cli[43] - for bundling css files
- myth[44] - a preprocessor for our css
- normalize-css[45] - reset the css for the browser
- skelestyle-typography[46] - a base set of typography css styles
- nodemon[47] - for running a development server

[35] http://localwiki.org
[36] http://SeattleWiki.net
[37] https://www.npmjs.org/package/hyperquest
[38] https://www.npmjs.org/package/director
[39] https://npmjs.org/package/handlebars
[40] https://npmjs.org/package/handlebars-layouts
[41] https://www.npmjs.org/package/st
[42] https://www.npmjs.org/package/event-stream
[43] https://www.npmjs.org/package/rework-npm-cli
[44] https://www.npmjs.org/package/myth
[45] https://github.com/sethvincent/normalize-css
[46] https://www.npmjs.org/package/skelestyle-typography
[47] https://www.npmjs.org/package/nodemon

Source code

You can find the full source code for this post here: github.com/learn-js/hyperquest-director-handlebars-example-app[48].

Basic usage examples

Here we'll take a look at some basic usage examples of each of the modules used in the application.

hyperquest

Hyperquest is a module for making streaming http requests.

Here's an example:

```
1  var request = require('hyperquest');
2
3  var req = request('http://localwiki.net/api/v4/pages/?region__slug=seattle&tags=\
4  pizza&format=json');
5  req.pipe(process.stdout);
```

This will pipe the response from the LocalWiki API to `process.stdout` (which console.log is an alias for in Node). Notice that we're requesting all pages from the `seattle` region tagged with `pizza` and asking for the response to be in the `json` format.

director

Director is used to handle routing requests in our server application. It can also be used for client-side and command line tools.

Here's an example of server-side usage of director:

[48] https://github.com/learn-js/hyperquest-director-handlebars-example-app

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

var port = process.env.PORT || 3000;
var router =   new director.http.Router();

var server = http.createServer(function(req, res){
  router.dispatch(req, res, function(err){
    if (err) {
      res.writeHead(404);
      res.end();
    }
  });
});

router.get('/', function(){
  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end('the root url');
});

router.get('/pizza', function(){
  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end('i like pizza');
});

router.get('/pizza/:adjective', function(adjective){
  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end('pizza is really ' + adjective);
});

server.listen(port);
console.log('app running on http://127.0.0.1:' + port);
```

Let's look at this example in chunks:

Require the `http` and `director` modules:

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

Set a port variable and instantiate the router object we'll use for routing requests:

```
var port = process.env.PORT || 3000;
var router = new director.http.Router();
```

Create a server, using `router.dispatch()` to handle requests:

```
var server = http.createServer(function(req, res){
  router.dispatch(req, res, function(err){
    if (err) {
      res.writeHead(404);
      res.end('not found');
    }
  });
});
```

If there's an error the browser will be sent a 404 message.

Set up a route for the route url:

```
router.get('/', function(){
  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end('the root url');
});
```

An example of an arbitrary route:

```
router.get('/pizza', function(){
  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end('i like pizza');
});
```

An example of using url parameters to alter responses:

```
router.get('/pizza/:adjective', function(adjective){
  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end('pizza is really ' + adjective);
});
```

Start the server, listen on the port in the `port` variable, and print a message to the console:

```
server.listen(port);
console.log('app running on http://127.0.0.1:' + port);
```

handlebars & handlebars-layouts

Handlebars is a common templating language. You can learn about its syntax and basic usage at handlebarsjs.com[49]. In this example we'll revise the director example to serve views compiled by Handlebars. We'll also use the handlebars-layouts module to allow for block layouts similar to those found in jade and django.

Here's the example:

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

var Handlebars = require('handlebars');
var hbsLayouts = require('handlebars-layouts')(Handlebars);

Handlebars.registerPartial('layout', fs.readFileSync('views/layout.html').toStri\
ng());
var template = Handlebars.compile(fs.readFileSync('views/index.html').toString()\
);

var site = {
  title: "Exampal usage of Handlebars",
  description: "Learn to use handlebars with node.js!"
}

var port = process.env.PORT || 3000;
var router = new director.http.Router();

var server = http.createServer(function(req, res){
  router.dispatch(req, res, function(err){
    if (err) {
      res.writeHead(404);
      res.end();
    }
  });
});
```

[49] http://handlebarsjs.com

```
29
30 router.get('/', function(){
31   var page = {
32     title: "This is the index page",
33     content: "This is the fornt page of the example handlebars site."
34   }
35
36   this.res.writeHead(200, { 'Content-Type': 'text/html' });
37   this.res.end(template({ site: site, page: page }));
38 });
39
40 server.listen(port);
41 console.log('app running on http://127.0.0.1:' + port);
```

And here's the example described in chunks:

Require the fs, http, and director modules:

```
1 var fs = require('fs');
2 var http = require('http');
3 var director = require('director');
```

Require handlebars and the handlebars-layouts modules:

```
1 var Handlebars = require('handlebars');
2 var hbsLayouts = require('handlebars-layouts')(Handlebars);
```

Create a partial using `Handlebars.registerPartial()` and create a template using `Handlebars.compile()`:

```
1 Handlebars.registerPartial('layout', fs.readFileSync('views/layout.html').toStri\
2 ng());
3 var template = Handlebars.compile(fs.readFileSync('views/index.html').toString()\
4 );
```

Create a `site` object with properties we'll use in the handlebars templates:

```
var site = {
  title: "Exampal usage of Handlebars",
  description: "Learn to use handlebars with node.js!"
}
```

Create a port variable and instantiate the router:

```
var port = process.env.PORT || 3000;
var router = new director.http.Router();
```

Create the server:

```
var server = http.createServer(function(req, res){
  router.dispatch(req, res, function(err){
    if (err) {
      res.writeHead(404);
      res.end();
    }
  });
});
```

Create a root route:

```
router.get('/', function(){
  var page = {
    title: "This is the index page",
    content: "This is the fornt page of the example handlebars site."
  }

  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end(template({ site: site, page: page }));
});
```

This creates a `page` object for use in the index template.

Note the following line:

```
this.res.end(template({ site: site, page: page }));
```

This uses the `template()` function that we created using `Handlebars.compile()`, and we're passing in the `site` and `page` objects for use in the template.

Start the server and print a message to the console:

```
server.listen(port);
console.log('app running on http://127.0.0.1:' + port);
```

The views used in this example

Here's the layout view:

```
<!doctype html>
<html lang="en-us">
<head>
  <title>{{site.title}}</title>
  <link rel="stylesheet" href="/static/bundle.css">
</head>
<body>

<header>
  <div class="container">
    <h1><a href="/">{{site.title}}</a></h1>
    <div>{{site.description}}</div>
  </div>
</header>

<main id="main-content" role="main">
  {% raw %}{{#block "body"}}{{/block}}{% endraw %}
</main>

</body>
</html>
```

This line: `{% raw %}{{#block "body"}}{{/block}}{% endraw %}` defines a block that we can override to place content into in views that use this one as a layout.

The index view:

```
{% raw %}
{{#extend "layout"}}

{{#replace "body"}}
<div class="container">
  <h1>{{page.title}}</h1>
  <div>{{page.content}}</div>
</div>
{{/replace}}

{{/extend}}
{% endraw %}
```

The `{% raw %}{{#extend "layout"}}{{/extend}}{% endraw %}` block allows this view to use the layout view as the layout.

Everything inside of this block: `{% raw %}{{#replace "body"}} ... {{/replace}}{% endraw %}` is rendered inside of the body block definition in the layout view.

st

st is a module for serving static files that pairs well with the director module.

Here's an example:

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

var staticFiles = st({ path: __dirname + '/static', url: '/' });

http.createServer(function(req, res) {
  if (staticFiles(req, res)) return
  else res.end('not a static file');
}).listen(3000);
```

Create a folder named static, and a file inside of it named example.txt. Put some text in that .txt file.

Now when you go to `http://localhost:3000/example.txt` that file will be rendered. Note that if you go to the root url with this set up you'll be shown the contents of the static directory. You can change that option by setting `index` to either false, or to a file that should be used as the index to the static files. Like this:

```
var staticFiles = st({ path: __dirname + '/static', url: '/', index: false });
```

event-stream

The event-stream module has a few uses, and in this case we'll be using its `wait()` method to wait until an end of a stream so that we can process the data that's coming in all at once.

Here's an example that integrates shows event-stream usage along with hyperquest:

```
var request = require('hyperquest');
var wait = require('event-stream').wait;

var api = 'http://localwiki.net/api/v4/';
var tag = 'pizza';
var pagesRequest = request(api + 'pages/?region__slug=seattle&tags=' + tag + '&f\
ormat=json');

pagesRequest
  .pipe(wait(function(err, data){
    var pages = JSON.parse(data).results;

    var pageTitles = '';
    for (var i=0; i<pages.length; i++){
      pageTitles += pages[i].name + ', ';
    }

    console.log('pages tagged with ' + tag + ': ' + pageTitles);
  }));
```

Here's the example broken into chunks and explained:

Require the hyperquest, combine-streams, and event-stream modules:

```
var request = require('hyperquest');
var wait = require('event-stream').wait;
```

Note that we're just using the `wait()` method from event-stream.

Set an `api` variable with the api url we'll be using and set the name of the tag we'll be requesting to `pizza`:

```
var api = 'http://localwiki.net/api/v4/';
var tag = 'pizza';
```

Make a request for all pages tagged with pizza:

```
var pagesRequest = request(api + 'pages/?region__slug=seattle&tags=' + tag + '&f\
ormat=json');
```

Pipe the response stream:

```
  pagesRequest
    .pipe(wait(function(err, data){
      var pages = JSON.parse(data).results;
```

The `data` argument we get back from the `wait` is a string of JSON that needs to be parsed into a JavaScript object. The `results` property is an array of pages.

Print the response to the console:

```
    var pageTitles = '';
    for (var i=0; i<pages.length; i++){
      pageTitles += pages[i].name + ', ';
    }

    console.log('pages tagged with ' + tag + ': ' + pageTitles);
  }));
```

For this example we're printing the names of all the pages that are tagged with pizza to the console.

rework-npm-cli & myth

rework-npm-cli is a command-line tool that uses rework-npm[50] to bundle css files that are packaged and distributed via npm. It's also useful for importing multiple local css files and bundling them into one file.

Basic usage of rework-npm-cli looks like this:

```
rework-npm source.css -o bundle.css
```

myth is a preprocessor for css that automatically adds prefixing for cross-browser support and provides polyfills for new CSS specs.

Basic usage of myth looks like this:

[50] https://npmjs.org/rework-npm

```
myth input.css output.css
```

Use them together by piping them like this:

```
rework-npm source.css | myth > bundle.css
```

normalize-css & skelestyle-typography

These two modules expose css files that we can import and bundle using the `rework-npm` command.

Install them like any npm module:

```
npm install --save normalize-css skelestyle-typography
```

These css modules each have a `style` property in their package.json files that determine what css we can use.

Import their css styles using the standard css `@import` statement:

```
@import "normalize-css";
@import "skelestyle-typography";
```

Then, using the `rework-npm` command, those files will be included into the bundle.css file:

```
rework-npm source.css -o bundle.css
```

nodemon

nodemon is a command-line tool for running node applications that restart when files are changed.

Basic usage is just replacing the `node` command with `nodemon`:

```
nodemon server.js
```

We'll look at a more complicated example at the end of this post.

Building the application

Now that we've run through the example usage of each module, building the actual application will almost be like review. We'll be plugging those modules together to form an application that grabs content from the SeattleWiki API, and serves that data to the browser as formatted HTML.

Setup the project

System dependencies

You'll need Node.js installed. For a guide to setting up a development environment, check out this post: Setting up a JavaScript / Node.js development environment[51].

Files and folders

Create a new project folder and navigate to it on the terminal:

```
mkdir example-project
cd example-project
```

Create folders for static files and views:

```
mkdir static views
```

Create the server.js file and the source.css file:

```
touch server.js source.css
```

Create layout, index, and page views in the views folder:

```
touch views/layout.html views/index.html views/page.html
```

Create a package.json file by running this command:

```
npm init
```

Answer the questions that it asks. Hit enter at a prompt to keep the default value.

Install dependencies

Install the development dependencies:

[51] http://learnjs.io/blog/2014/01/22/js-development-environment/

```
npm install --save-dev rework-npm-cli myth nodemon
```

The `--save-dev` option saves the modules and their current version numbers to the `devDependencies` field in the package.json file.

Install the project dependencies:

```
npm install --save hyperquest director handlebars handlebars-layouts st event-st\
ream skelestyle-typography normalize-css
```

The `--save` option saves the modules and their current version numbers to the `dependencies` field in the package.json file.

Your `dependencies` and `devDependencies` fields should now look like this:

```
"dependencies": {
  "director": "~1.2.2",
  "event-stream": "~3.1.0",
  "handlebars": "~1.3.0",
  "handlebars-layouts": "~0.1.3",
  "hyperquest": "~0.2.0",
  "normalize-css": "^2.3.1",
  "skelestyle-typography": "0.0.4",
  "st": "~0.2.5"
},
"devDependencies": {
  "rework-npm-cli": "0.0.1",
  "myth": "~0.3.0",
  "nodemon": "~1.0.14"
}
```

Create the server

First we'll add the server code to the server.js file. This mostly compiles previous examples that we've shown into one full file.

Here's the full server.js file:

```
var fs = require('fs');
var http = require('http');
var st = require('st');
var director = require('director');
var request = require('hyperquest');
var wait = require('event-stream').wait;
var Handlebars = require('handlebars');
var hbsLayouts = require('handlebars-layouts')(Handlebars);

var wiki = {
  name: 'Seattle LocalWiki',
  url: 'http://localwiki.net/seattle',
  api: 'http://localwiki.net/api/v4/'
}

Handlebars.registerPartial('layout', fs.readFileSync('views/layout.html').toStri\
ng());

var templates = {
  index: getView('index'),
  page: getView('page')
}

var port = process.env.PORT || 3000;
var router =   new director.http.Router();
var staticFiles = st({ path: __dirname + '/static', url: '/static', passthrough:\
 true })

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

  /*
  * if the request is for a static file, handle it here
  */
  if (staticFiles(req, res)) return;

  /*
  * otherwise, let the router handle the request
  */
  router.dispatch(req, res, function(err){
    if (err) {
      res.writeHead(404);
      res.end();
```

```
    }
  });
});

router.get('/', function(){
  var html = templates.index({ wiki: wiki });
  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end(html);
});

router.get('/:id', function(id){
  var self = this;
  var pagesRequest = request(wiki.api + '/pages/?region__slug=seattle&tags=' + i\
d + '&format=json');

  pagesRequest
    .pipe(wait(function(err, data){

      var html = templates.page({
        wiki: wiki,
        tag: id,
        pages: JSON.parse(data).results
      });

      self.res.writeHead(200, { 'Content-Type': 'text/html' });
      self.res.end(html);
    }));
});

server.listen(port);
console.log('app running on http://127.0.0.1:' + port);

/*
* helper function for pulling in a handlebars template
*/
function getView(file){
  return Handlebars.compile(fs.readFileSync('./views/' + file + '.html').toStrin\
g());
}
```

Here's the server.js file broken into chunks and explained:

Require the project dependencies:

```
var fs = require('fs');
var http = require('http');
var st = require('st');
var director = require('director');
var request = require('hyperquest');
var wait = require('event-stream').wait;
var Handlebars = require('handlebars');
var hbsLayouts = require('handlebars-layouts')(Handlebars);
```

Create a wiki object that will be used in templates:

```
var wiki = {
  name: 'Seattle LocalWiki',
  url: 'http://localwiki.net/seattle',
  api: 'http://localwiki.net/api/v4/'
};
```

Register the layout partial:

```
Handlebars.registerPartial('layout', fs.readFileSync('views/layout.html').toStri\
ng());
```

Use the `getView()` function to create template functions we can use later:

```
var templates = {
  index: getView('index'),
  page: getView('page')
};
```

We'll look at the `getView()` helper function later in the code.

Create the port variable, and instantiate router and staticFiles response handler:

```
var port = process.env.PORT || 3000;
var router =  new director.http.Router();
var staticFiles = st({ path: __dirname + '/static', url: '/static', passthrough:\
 true });
```

Create the server:

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

  /*
  * if the request is for a static file, handle it here
  */
  if (staticFiles(req, res)) return;

  /*
  * otherwise, let the router handle the request
  */
  router.dispatch(req, res, function(err){
    if (err) {
      res.writeHead(404);
      res.end();
    }
  });
});
```

The `staticFiles` response handler takes care of a request if it matches any of the available static files. Otherwise the `router` handler takes care of the request.

Create the root route:

```
router.get('/', function(){
  var html = templates.index({ wiki: wiki });
  this.res.writeHead(200, { 'Content-Type': 'text/html' });
  this.res.end(html);
});
```

Create the page route:

```
router.get('/:id', function(id){
  var self = this;
  var pagesRequest = request(wiki.api + '/pages/?region__slug=seattle&tags=' + i\
d + '&format=json');
```

We use `id` as a parameter that's used in the requests we send to the wiki API.

** Request the pages from the API, use the `template.page()` function to build the HTML that's sent to the browser:**

```
  pagesRequest
    .pipe(wait(function(err, data){

      var html = templates.page({
        wiki: wiki,
        tag: id,
        pages: JSON.parse(data).results
      });

      self.res.writeHead(200, { 'Content-Type': 'text/html' });
      self.res.end(html);
    }));
});
```

Like in the event-stream example above, this section takes the string we get in the `data` argument, parses the JSON, and is added to the object along with the `wiki` that's passed into the `templates.page()` function to build the HTML that's sent to the browser.

Start the server and print a message to the console:

```
server.listen(port);
console.log('app running on http://127.0.0.1:' + port);
```

Create helper function that reads an HTML file to compile a Handlebars template:

```
/*
* helper function for pulling in a handlebars template
*/
function getView(file){
  return Handlebars.compile(fs.readFileSync('./views/' + file + '.html').toStrin\
g());
}
```

Create the project views

The views for this project are very similar to the examples we used above when describing the use of Handlebars.

The layout.html view:

```
<!doctype html>
<html lang="en-us">
<head>
{{#block "head"}}
  <title>{{ wiki.name }}</title>
  <link rel="stylesheet" href="/static/bundle.css">
{{/block}}
</head>
<body>

<header>
  <div class="container">
    <h1><a href="/">Pages on {{ wiki.name }}</a></h1>
  </div>
</header>

<main id="main-content" role="main">
  {{#block "body"}}{{/block}}
</main>

</body>
</html>
```

The index.html view:

```
{{#extend "layout"}}

{{#replace "body"}}
<div class="container">
  <p>Check out some of the pages on the {{ wiki.name }}!</p>
  <div id="tags">
    <h2>Here are some examples:</h2>
    <ul>
      <li><a href="/wallingford">Wallingford</a></li>
      <li><a href="/pizza">Pizza</a></li>
      <li><a href="/pioneersquare">Pioneer Square</a></li>
    </ul>
  </div>
</div>
{{/replace}}

{{/extend}}
```

The page.html view:

```
{{#extend "layout"}}

{{#replace "body"}}
<div class="container">
  <h1>{{ tag.name }}</h1>
  <p>All pages on the <a href="{{ wiki.url }}">{{ wiki.name }}</a> tagged with <\
b>{{ tag }}</b>.</p>
  <div id="pages">
    {{#each pages}}
    <h2><a href="{{ ../wiki.url }}/{{ name }}" target="_blank">{{name}}</a></h2>
    {{/each}}
  </div>
</div>
{{/replace}}

{{/extend}}
```

Creating the source.css file

I'll keep the css simple. Just something to show how rework-npm-cli works for bundling css.

Add this to your source.css file:

```
@import "normalize-css";
@import "skelestyle-typography";

.container {
  width: 70%;
  margin: 0px auto;
}
```

Feel free to add whatever css rules you like to improve the style of the project.

Next we'll look at the commands needed to bundle the css and start a development server.

Create npm scripts in your package.json file

We'll need three npm scripts:

- One for bundling the css.
- One for watching the css files for changes and rebundling the css

- One for starting the server.

You'll add each one of these to the `scripts` field in the package.json file:

```
"scripts": {

},
```

Bundling the css:

```
rework-npm source.css | myth > static/bundle.css
```

This will create the bundle.css file that imports the css styles from normalize-css and skelestyle-typography.

We'll add this as an npm script called `bundle-css`:

```
"bundle-css": "rework-npm source.css | myth > static/bundle.css"
```

It can now be run on it's own with `npm run bundle-css`.

Watching the css for changes:

```
nodemon -e css --ignore static/* --exec 'npm run bundle-css'
```

This uses npm to watch the css file for changes, ignoring everything in the static folder, and executes the bundle-css script on each change.

We'll call this script `watch-css`:

```
"watch-css": "nodemon -e css --ignore static/* --exec 'npm run bundle-css'"
```

It can now be run on it's own with `npm run watch-css`.

Running the development server

```
nodemon -e js,html server.js & npm run watch-css
```

This nodemon command watches JavaScript and HTML files for changes and will restart the server. It also runs the `watch-css` command to kick it off and watch for css changes.

Let's use it as the `start` script:

```
"start": "nodemon -e js,html server.js & npm run watch-css"
```

It can now be run with `npm start`.

The `scripts` field in te package.json file should now look like this:

```
  "scripts": {
    "bundle-css": "rework-npm source.css | myth > static/bundle.css",
    "watch-css": "nodemon -e css --ignore static/* --exec 'npm run bundle-css'",
    "start": "nodemon -e js,html server.js & npm run watch-css"
  },
```

The command `npm start` will get your server running, and you'll be able to access the site at http://localhost:3000.

Preloading and rendering image sprites in 2d js/canvas games

Learn about using the sprite-2d[52], load-images[53], and gameloop[54] modules to create animations for 2d canvas/javascript games.

We'll also use the beefy[55] and browserify[56] modules as development tools.

Set up the project

> For help getting started with Node.js/JavaScript development environments, check out this blog post: Setting up a JavaScript / Node.js development environment[57].

Create and change directory into a new project folder:

```
mkdir my-project-name
cd my-project-name
```

Now create a package.json file using `npm init`

```
npm init
```

This command will ask you questions about your project. Answer all the questions and it'll create a package.json file for you.

Install dependencies

Install gameloop, load-images, and sprite-2d:

[52] http://github.com/sethvincent/sprite-2d
[53] http://github.com/sethvincent/load-images
[54] http://github.com/sethvincent/gameloop
[55] https://github.com/chrisdickinson/beefy
[56] https://github.com/substack/browserify
[57] http://learnjs.io/blog/2014/01/22/js-development-environment/

```
npm install --save gameloop load-images sprite-2d
```

Install beefy and browserify as development dependencies:

```
npm install --save-dev beefy browserify
```

gameloop

The gameloop module creates the bare-bones states and events for a game: start, end, update, draw, pause, resume.

Here's an example:

```
var Game = require('gameloop');

var canvas = document.createElement('canvas');
document.body.appendChild(canvas);

var game = new Game({
  renderer: canvas.getContext('2d')
});

game.width = canvas.width = 800;
game.height = canvas.height = 400;

game.on('start', function(){
  console.log('started', this);
});

game.on('update', function(dt){

});

game.on('draw', function(context){
  context.clearRect(0, 0, game.width, game.height);
});

game.start();
```

load-images

The load-images module does one job: loading images.

Basic usage looks like this:

```
loadImages(arrayOfFilePaths, function(err, images){
  console.log(err, images);
  game.images = images;
  game.start();
});
```

Here's the gameloop example extended to include usage of the load-images module:

```
var Game = require('gameloop');
var loadImages = require('load-images');

var canvas = document.createElement('canvas');
document.body.appendChild(canvas);

var game = new Game({
  renderer: canvas.getContext('2d')
});

game.width = canvas.width = 800;
game.height = canvas.height = 400;

loadImages(['zombie-baby.png', 'zombie-baby2.png'], function(err, images){
  console.log(err, images);
  game.images = images;
  game.start();
});

game.on('start', function(){
  console.log('started', this);
});

game.on('update', function(dt){

});

game.on('draw', function(context){
  context.clearRect(0, 0, game.width, game.height);
  context.drawImage(game.images['zombie-baby.png'], 50, 50);
  context.drawImage(game.images['zombie-baby2.png'], 100, 100);
});
```

sprite-2d

The sprite-2d module does the work of looping through frames in a spritesheet to animate sprites, likely most useful for 2d canvas games.

Basic usage looks like this:

```
var Sprite = require('sprite-2d');

var sprite = new Sprite({
  image: image,
  frames: 4,
  fps: 20
});
```

In your update loop:

```
sprite.update(dt);
```

This advances the sprite to the next frame based on the fps specified when creating the sprite.

In your draw loop:

```
sprite.draw(function(image, frame){
  context.drawImage(
    image,
    frame.position,
    0,
    frame.width,
    image.height,
    entity.position.x,
    entity.position.y,
    frame.width,
    image.height
  );
});
```

The `sprite.draw` method takes a callback that provides two arguments: `image` which is the actual image used for the sprite, and `frame`, which represents the current frame of the spritesheet.

These two arguments give us everything we need to draw the sprite using something like the `context.drawImage` method, like seen above.

Here's our example extended to integrate the sprite-2d and load-images modules:

```
var Game = require('gameloop');
var loadImages = require('load-images');
var Sprite = require('sprite-2d');

var canvas = document.createElement('canvas');
document.body.appendChild(canvas);

var game = new Game({
  renderer: canvas.getContext('2d')
});

game.width = canvas.width = 800;
game.height = canvas.height = 400;

var sprite;

loadImages('zombie-baby.png', function(err, images){
  if (err) throw err;

  sprite = new Sprite({
    game: game,
    image: images['zombie-baby.png'],
    frames: 4,
    fps: 8
  });

  game.start();
});

game.on('start', function(){
  console.log('started', this);
});

game.on('update', function(dt){
  sprite.update(dt)
});

game.on('draw', function(context){
  context.clearRect(0, 0, game.width, game.height);
```

```
  sprite.draw(function(image, frame){
    context.drawImage(
      image,
      frame.position,
      0,
      frame.width,
      image.height,
      100,
      100,
      frame.width,
      image.height
    );
  });
});
```

To get this to run you'll need an image in the root folder of your project named 'zombie-baby.png'. You can use this one if you want: https://github.com/sethvincent/hogjam4/blob/gh-pages/images/zombie-baby.png[58]

Running the example

Put the above example code into a file named `game.js`.

Use this command to run a local development server with beefy:

```
beefy game.js:bundle.js --live
```

[58] https://github.com/sethvincent/hogjam4/blob/gh-pages/images/zombie-baby.png

More on the way!

More tutorials

Expect this book to continue to be updated over the next year with more tutorials!

Interested in specific modules from npm? Email hi@learnjs.io with your suggestions, or create an issue on the project issue queue on GitHub: github.com/learn-js/npm-recipes/issues[59].

More JavaScript books

There's a good chance that if you like this book you'll be interested in the other books in the Learn.js series.

Check them out!

- Introduction to JavaScript & Node.js[60]
- Making 2d Games with Node.js & Browserify[61]
- Mapping with Leaflet.js[62]
- Development Environments for Beginners[63]
- Theming with Ghost[64]

Learn more at learnjs.io[65].

[59] http://github.com/learn-js/npm-recipes/issues
[60] http://learnjs.io/books/learnjs-01
[61] http://learnjs.io/books/learnjs-02
[62] http://learnjs.io/books/learnjs-03
[63] http://learnjs.io/books/dev-envs
[64] http://themingwithghost.com
[65] http://learnjs.io

Changelog

v0.2.1 - September 7, 2014

- Fix hyperquest chapter to use latest localwiki.net api, and other small fixes

v0.2.0 – March 4, 2014

- Add chapter 4: Preloading and rendering image sprites in 2d js/canvas games

v0.1.0 - February 11, 2014

- Add introduction
- Add chapter 1: creating js library builds with browserify, watchify, and uglify-js
- Add chapter 2: rework-npm, myth, & clean-css
- Add chapter 3: hyperquest, director, handlebars, and more