roots Documentation

Release 3.0.0

jeff escalante

March 10, 2014

Roots is a fast, simple, and customizable static site compiler.

CHAPTER 1

Table of Contents

1.1 Installation

It's quite straightforward to install roots. First, make sure that node js has been installed, then run:

```
npm install roots -g
```

If you experience any errors, please file an issue and we'll try our best to make things work for you!

1.2 Features

There are lots of other static compilers on the market. I'd like to explain why I continue working on roots depsite this, and eventually this document will also probably contain a feature comparison between a few of the more popular ones.

1.2.1 Motivation

Roots is essential to my and my company's daily work. The work I do on my own is freelance, and the work I do at my company is very similar, which means that rather than working on a single project, I tend to move very quickly between a lot of different projects with a lot of different people. This means that I need a very strong and flexible system that is actively maintained with a clean and extensible codebase in order to be confident that any project can be handled.

In addition, when working on lots of web projects quickly, it becomes more and more important to wrap up common patterns and eliminate unnecessary time sinks. This often means writing on top of languages that compile down to html, having a wide range of compilers supported, and having those compilers be flexible to different options, a core piece of roots.

Finally, it's important to accomplish all these goals with the least amount of configuration code, and the most clear documentation possible. Often times other people will jump in and out of projects, and it's important that anyone is able to quickly get a handle on the code and get things up and running fast. Roots is the only system I am aware of that satisfies all these requirements.

1.2.2 Feature List

- speed
- custom compiler options
- before/after hooks

- precompiled templates
- dynamic content
- new project templating
- simple deployment
- client-side javascript management
- multipass compilation
- live reload in browser
- clean and clear error handling

1.3 Roots Config

You can configure roots through an optional `app.coffee` file at the root of your project. Although it is not required for simple projects, there are a lot of very powerful options you can take advantage of, which are explained below in the options section. But first, let's talk about the format of the file

1.3.1 Format

`app.coffee` can come in two flavors. The first is more simple, just configured as a coffeescript object. For example:

```
output: 'public'
env: 'development'
after: -> console.log('what a useful function')
```

This is a great way to format the file for maximum simplicity. It ends up being very clean and easy to manage. However, if you want to do more advanced things like `require``ing in files node/commonjs-style, you will not be able to do this. It is parsed simply as an object, not with full node functionality. If you do want full node functionality though, you're in luck -- all you have to do is add ``module.exports =` to the top, like this:

```
axis = module.require('axis-css')
autoprefixer = module.require('autoprefixer-stylus')

module.exports =

  output: 'public'

  stylus:
    use: [axis, autoprefixer]
```

As you can see, here we are able to locally require in extra dependencies and push them directly into the roots pipeline. Make sure if you are requiring locally to note the use of `module.require` - since this is loaded into roots' context, you'll need the `module` prefix in order to load your deps from the right place.

1.3.2 Options

Below are all the options that you can pass to roots, from the simplest to the most advanced.

output

The path to a folder (starting from project root) that your project will be compiled into. *default: ``public``*

ignores

An array containing minimatch strings that represent files or folder you wish to ignore from the compile process. Full globstar syntax supported. Automatically ignores `package.json`, `node_modules`, `app.coffee` and your output directory (wouldn't want to have it recursively compile itself!)

dump_dirs

Array of directories that will have their contents dumped into the output folder rather than compiling into the folder they are in. *default: ''['views', 'assets']''*

env

Basic environment variable. Usually set through command line options, but if you need you can override here. *default: ''development''*

debug

When enabled, commands will dump out lots of information on what roots is doing internally. *default: ''false''*

live_reload

When enabled, on *roots watch*, the browser will automatically reload every time you save a file in your project. *default: ''true''*

open_browser

When enabled, *roots watch* will automatically open a browser to the local server. *default: ''true''*

locals

An object that is injected into the options every compiler in use in the project. So for example, if you are using both jade and ejs in a project and some locals to be the same across the two, you don't have to duplicate, just add them to `locals`. If there is a conflict between `locals` and compiler-specific options, the compiler options will win out.

before

Hook function that is run before each compile. Function passes in an instance of the roots class, so you have access to everything. Accepts either a single function or an array of functions, which will be run in order. Expects a promise or value to be returned from each function.

after

Same thing as before, but is run after each compile. Surprise surprise.

1.3.3 Compiler Options

You can also pass options directly to any compiler through `app.coffee`. Just add them as an object under the name of the compiler. For example, if you want jade to output non-compressed html:

```
jade:
  pretty: true
```

That's all it takes. This will work for any compiler you have loaded. For more info on each supported compiler's options, see the accord docs.

1.4 Errors

Unfortunately, errors happen. If one does occur, it might be our fault and it might be your fault. When roots throws an error that we have recognized is possible, we try to throw it with as clear and human-readable an error message as

possible to make it easy to see where things went downhill. Documented here are all the error codes that roots can throw. It's certainly possible that there's an error that doesn't match any of these, if so please file an issue!

The numbers you see here are unix error codes. They start at *125* because this is where the standard error codes end. If you are using roots programatically, expect for the program to exit with the same code as is documented here.

1.4.1 125 - Malformed Extension

This error means that you fed roots an extension of the incorrect type. To fix this, check through your extensions and make sure that they all are correctly formatted according to the roots extension spec.

1.5 Roots Extension API

If there is more functionality you want to add to roots, you can probably do this with a plugin. There are a number of plugins that are officially maintained:

- roots-dynamic-content
- roots-precompiled-templates
- roots-js-pipeline
- roots-css-pipeline
- roots-browserify
- roots-json-content

There are also plenty of extensions that are not officially maintained and are still awesome. We will soon have a directory listing of these on the roots website for your sorting and browsing pleasure.

Documentation for building your own extensions can be found below.

1.5.1 Building an Extension

Roots extensions are extremely powerful, and have the ability to transform roots into more or less anything you want. A lot of what are currently extensions used to be written directly to the core, integrated throughout it in multiple places. Therefore, the extensions API has hooks into many different places in roots' core compile pipeline, and to understand how to write an extension, it's important to understand at least in general how roots works.

Let's start at the beginning. When roots starts compiling your project, it scans the folder for all files, and sorts them into categories. By default, it will sort files into *compiled* or *static*, with the compiled files being ones that match file extensions of compilers that you have installed, and static being files that should simply be copied over. *This is the first place that your extension can jump in*.

Extensions export a function that returns a class. The function is executed when roots is initialized, this is the time for setting up any global options or configuration that persist across as many compiles as will happen. The extension must then export a "class" (which in js is really a function, but since these examples use coffeescript we'll go with class). This class will be *re-instantiated each time compile is run*. This means that any context inside the class is cleared between compiles. If there is anything you want to be able to hold on to between compiles, it needs to be outside the class. Conversely, do not store anything outside the class that you do not want to be untouched between two compiles. For example, if you add content to an array outside each compile, it will continue getting larger each compile. This is usually not what you want to do, so assume that anything outside the class is used for global configuration for the extension. With that out of the way, let's lay down the skeleton for a sample extension that finds any file with a filename in all caps and makes sure the contents are also all caps.

tl;dr - your extension must be a function that returns a function:

```
module.exports = (opts) ->

  # any initialization code here

  class YellExtension
    constructor: (@roots) ->
```

Notice that an instance of the roots class is passed into the constructor. This instance will give you access to literally any and all information that roots is hanging on to. You can grab things out of the app.coffee file, you can access compile adapters, etc. And this is not a clone of the instance, this is *the* instance, so modifying values could screw things up. Weild this repsonsibility carefully if you choose to at all. Since we don't need it for this extension, the constructor will be omitted from the rest of the example code in this guide.

1.5.2 File Sorting

Ok, so we have a start. Now, in order to get into the filesystem scanning portion, we want to define a category that we'll sort the targeted files into, as well as a function that we can use to detect whether this is a file we want to separate into our own category, which, for this extension, means that it's filename will be in all uppercase. We can do this by defining a *fs* method on the class which returns an object with *category* and *detect* properties:

```
path = require 'path'

module.exports = (opts) ->

  class YellExtension

    fs: ->
      category: 'upcased'
      detect: (f) ->
        path.basename(f.relative) == path.basename(f.relative).toUpperCase()
```

So category is just a string (we can use this later), and detect is a function which is fed a vinyl wrapper for each file that's run through. Here, we just run a simple comparison to see if the basename is all uppercase. The *detect* function also can return a promise if you are running an async operation. Do note that speed is important in roots, so make sure you have considered the speed impacts of your extension. That means try not to for example read the full contents of a file synchronously, because that could take quite a while in a larger project.

There are a couple more options to consider here in the filesystem sorting section. First, it's possible that multiple extensions could be operating on the same project, and it's important to consider the order in which they run, and whether files are "caught" by one extension or passed through to others. You can handle this with the *extract* boolean, which can be set to *true* in order to stop the file from being potentially sorted into other categories after detection. In this case we do want that, since we want the file to be compiled *only* as all uppercase, not also compiled normally after. This is the case for most extensions. Let's update our code:

```
path = require 'path'

module.exports = (opts) ->

  class YellExtension

    fs: ->
      category: 'upcased'
      extract: true
      detect: (f) ->
        path.basename(f.relative) == path.basename(f.relative).toUpperCase()
```

Finally, it's possible that you actually need your category to be compiled **before** anything else compiles. For example, dynamic content is compiled before anything else, because it makes locals available to all other view templates. Since roots compiles all files as quickly as possible, compiling dynamic content alongside normal views would result in race conditions where only some dynamic content would be available in the rest of the views. For that reason, the extension must ensure that the entire "dynamic" category is finished compiling before the rest of the project begins. This of course has speed implications as well which should be considered, but if it's necessary, it's necessary.

For this extension, there's no need for the file to be compiled before others, so we can skip the *ordered* property, which defaults to *false*. And that will do it for the filesystem sorting portion, we now have a neat list of all files with upcased filenames and are ready to move on to the compile hooks, where we get a chance to modify the content.

1.5.3 Compile Hooks

The next step for us is to modify the file's content. A good way to do this would be to snag a hook after the file is finished compiling, but before it is written, that upcases all the content. Luckily, we can easily do this as such:

```
path = require 'path'

module.exports = (opts) ->

  class YellExtension

    fs: ->
      category: 'upcased'
      extract: true
      detect: (f) ->
        path.basename(f.relative) == path.basename(f.relative).toUpperCase()

    compile_hooks: ->
      after_file: (ctx) =>
        if ctx.category == @fs.category
          ctx.content = ctx.content.toUpperCase()
```

So let's talk about this. First, we have the `compile_hooks` method, which returns an object with 4 potential hooks, one that we've seen: `before_file`, `after_file`, `before_pass`, and `after_pass`. The "pass" hooks fire once for each compile pass taken on the file (files can have multiple extensions and be compiled multiple times), and the "file" hooks fire once per file, no matter how many extensions it has or how many times it is compiled. Each hook is passed a context object, which is an instance of a class. The file hooks get an instance of the CompileFile class, and the pass hooks get the CompilePass class. The information available in each class will be listed in the next section.

After this hook, the file goes on to be written, and all is well! Only one caveat, if you return false or a promise for false from the `after_file` hook, the file **will not be written**.

1.5.4 Information Available to Compile Hooks

You can get at and/or change any piece of data that roots holds on to through the `ctx` objects passed to the compile hooks, making them very powerful. The object is arranged such that the information you probably need is easiest to get to. We'll go through the object level by level.

1.5.5 "File" Hooks

- `roots`: roots base class instance, holds on to all config info
- `category`: the name of the category that the file being compiled is in

- `path`: absolute path to the file
- `adapters`: array of all accord adapters being used to compile the file
- `options`: options being passed to the compile adapter
- `content`: self-explanitory

1.5.6 "Pass" Hooks

- `file`: the entire object documented directly above this
- `adapter`: the accord adapter being used to compile the current pass
- `index`: the number of the current pass
- `content`: self-explanitory

1.5.7 Category Hooks

There is one more hook you can use that will fire only when all the files in a given category have completed processing. You can define one as such:

```
module.exports = (opts) ->

  class FooBar

    category_hooks: ->
      after: (ctx, category) ->
        console.log "finished up with #{category}!"
```

This is all pretty straightforward stuff. Example usage could be if you wanted to stop the write for all files in your category, then manually write them once the whole category is finished, maybe to just one file. the `ctx` object is slightly less interesting this time although it does still contain the `roots` object with access to all the settings you need.

1.5.8 Write Hook

You can also hook into the method that writes files in roots and use it to write more than one file. Under `compile_hooks`, if you add a `write` method, it will allow you to jump in. The write hook expects a specific output and *if you do not provide this output, it will crash*, so take note. From the write hook, you must return either an object or an array of objects that have two keys:

- `path`: the absolute path to where the file should be written
- `content`: the content you want to write to the file

So if you want to write multiple files out of one input, you can just override the write hook, do your path and content figuring, and return an array, one object for each file you want to write. Note that you can also return a promise for your object or array of objects if you need to do async tasks here.

You have access to a full context object from the write hook, as with anything else. The context in this hook is an exact mirror of the context that you get in the after file hook. Finally, if you return false out of the write hook, nothing will be written, as is the case with the after hook.

1.5.9 Adding an Extension to Roots' Pipeline

Adding an extension to roots is fairly simple. All you have to do is add it to an `extensions` array in `app.coffee`. For example:

```
yell = module.require('yellr')

module.exports =

  extensions = [yell()]
```

So what's happening here is that we assume that we have `npm install''ed our extension, ''yellr` locally. We then use `module.require` to require it from local (since this file is evaluated inside roots, using `module.require` ensures that the require root is from your project's folder), call it to initialize, and add it to the extensions array. If there were any options for the plugin, they would be passed in on this initialization.

This call returns the extension's class, and a fresh instance of the class is initialized each compile pass. This way, you can hold on to "global" extension config passed in through the function wrapper, but you don't get any overlap or confusion between each compile pass.