roots Documentation

Release 3.0.0

jeff escalante

March 21, 2014

Contents

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@pre -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 site generators. This page will attempt to explain what keeps us working hard on roots.

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 and in-browser notification

1.2.3 Comparisons

There are boatloads of other static site genrators out there, and here we'll analyze a couple of them briefly, what they are best for, and how they compare to Roots.

- **DocPad**: Powerful, but complex interface for users and developers. Roots has the same power, and is complex for developers, but simple for users.
- **Grunt**: General purpose build tool that has everything you need, but a very ugly interface and everything needs to be wired together manually. Roots has a very slim and clean interface and comes pre-wired specifically for building websites.
- **Jekyll**: Very blog-specific, slower because ruby, very difficult to extend with different languages, plugins and workflows. Roots is significantly faster, more flexible with the type of site you can build, and more extensible.
- **Octopress**: Very similar to jekyll, except even more specific to blogs. Again, Roots is a lot more flexible in this regard.
- **Middleman**: Really an excellent and full-features static generator if you are into ruby. Roots' only advantage here is speed, works better with js languages, and multipass compiles.
- **Wintersmith**: Complex interface for users and developers - users are required to shape and interace directly with the file/parse tree. Roots insulates users from this and exposes a much simpler interface.
- **Metalsmith**: Much like grunt or gulp, this is a build tool that can be configured to make static sites. Roots is a tool specifically for building static sites, and comes with a bunch of extra features and conveniences.
- **Hexo**: Blog-specific. Roots is more flexible than this.
- **Stasis**: Little known fact – the first version of roots was actually just stasis with some css libraries. I love stasis, but eventually outgrew it's limited feature set. It also appears to be unmaintained now.
- **Nanoc**: TODO
- **Pelican**: TODO
- **Harp**: TODO
- **Punch**: TODO
- **Brunch**: TODO

In my very humble opinion, the strongest alternatives to roots are, in order, **Middleman**, **Brunch**, **Docpad**, and **Metalsmith**. That is for some reason if you have spent all this time here and ended up deciding that you actually don't like Roots.

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 the specific error message that roots gave you, read through the extension docs linked above, and make sure that everything is formatted correctly. If you are having trouble writing an extension, feel free to drop in to the roots support channel

1.4.2 126 - Malformed Write Hook Output

This error means that your extension's write hook returned incorrectly formatted output. See the docs for write hooks (linked above) for more details on how to correct this.

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-component
- roots-json-content
- roots-i18n

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.

1.5.1 Adding an Extension to Roots' Pipeline

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

```
example = module.require('example')

module.exports =

  extensions = [example({ option: true })]
```

So what's happening here is that we assume that we have `npm install''ed our extension, ''example` 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.

You can add multiple extensions to this array of course. Each extension should have it's own documentation on options that can be passed and how it can be installed.

1.5.2 Building an Extension

This is quite a long piece, and due to roots' core complexity and the desire to make extensions powerful enough to accomplish just about anything you'd need, **understanding how to build an extension is not easy**. However, if you take the time to really read this through, all the concepts are explained thoroughly, and if you need help with anything, someone on the roots core team can help you out here.

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. We will be discussing this throughout the document.

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*.

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.

```
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.

1.5.3 File Sorting

Now let's talk about the first spot that we can jump into the roots pipeline. 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` categories, 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 without modification. *This is the first place that your extension can jump in*.

The bare basics we want to have defined here are a category that we'll sort the targeted files into, as well as a function that we can use to actually do the sorting. For this extension, this will just be a function that detects whether a filename is in all uppercase characters. 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 = ->
  class YellExtension

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

So the 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 it's likely that your `detect` function will be run for most or all files in a project, 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 with lots of files.

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. To be clear, if you were to not set the `extract` property, which is `false` by default, if a file was "detected" by our custom extension, it would be sorted into that category, but also able to be sorted into any other extension that operates after it as well. Setting `extract` to true ensures that once your extension detects a file, it is not able to be added to any other extension's category. Let's update our code to make sure that it works the way we want:

```
path = require 'path'

module.exports = ->
  class YellExtension

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

Much better. Now, it's also 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. To make it such that your extension's category is *finished processing before any other category starts*, you can set the `ordered` property on `fs` to `true`. If there are multiple extensions for which this is the case, the order in which they will run matches the order that they are added to the roots project, aka their order in the `extensions` array in the `app.coffee` file.

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 our extension producing 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.

1.5.4 Compile Hooks

The next step for us is to modify the file's content to actually make it all uppercase. A good way to do this would be to use 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 = ->
  class YellExtension

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

    compile_hooks: ->
      category: 'upcased'
      after_file: (ctx) =>
        ctx.content = ctx.content.toUpperCase()
```

So let's talk about this. First, we have the `compile_hooks` method, which returns an object with 5 potential hooks, one that we see used here. The five hooks you can add to `compile_hooks` are:

- `before_file`
- `after_file`
- `before_pass`
- `after_pass`
- `write`

The "pass" hooks fire once for *each compile pass taken on the file* (files can 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 (seen as `ctx` in the example above), 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 below.

Using the context, we can modify more or less anything. In the example above, we modify the content and change it to be an upcased version of the content. After this, the file moves on to be written!

1.5.5 Categories

Before we jump in to the next section, let's take a moment to talk about the `category` property. This is a piece of information that can be used in a few different sections, so it's a bit more flexible. You might have noticed a little bit of redundancy in the previous example, which we can eliminate here with clever use of the `category` property.

If you define a `category` on the class itself, that category is automatically applied to all hooks. Let's use this to do a quick refactor of the code above.

```
path = require 'path'

module.exports = ->
  class YellExtension
    constructor: ->
      @category = 'upcased'

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

    compile_hooks: ->
      after_file: (ctx) =>
        ctx.content = ctx.content.toUpperCase()
```

See the difference? Adding `@category` to the class itself has allowed us to remove the `category` property on both the `fs` and `compile_hooks` blocks. Now all the repetiton has been cut, awesome! In addition, if you have a class-level category as above and want to override one specific set of hooks with a *different* category, you can always define it explicitly on the set of hooks (like we did previously), and it will override the category on the class. For example:

```
path = require 'path'

module.exports = ->
  class YellExtension
    constructor: ->
      @category = 'unused-category'
```

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

compile_hooks: ->
  category: 'upcased'
  after_file: (ctx) =>
    ctx.content = ctx.content.toUpperCase()
```

While this example makes no sense in reality, it demonstrates the use of category overrides correctly. The class-level category here is overridden in every method by another category. If for some reason you want to just override it in some methods, you can do that as well.

Finally, if you *don't define a category at all* (and you don't have a `fs` method, which relies on having a category, *or* the `fs` method has it's own scoped category), your hooks will run for *all categories*.

1.5.6 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 a little more difficult to understand, but 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.

1.5.7 "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

1.5.8 "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

1.5.9 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 = ->
  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. The `category_hooks` method is scoped to categories in the same way that every other hook is. You can define it with a class-level category, a `category` property on the object it returns, or not use a category at all and have it run for every category.

1.5.10 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 a boolean, an object, an array of objects, or a promise for any of these values. Any object returned can have two keys:

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

Let's go through what each return value will accomplish. If you return `true`, the file will be written as usual, as if the write hook never intervened. If you return `false`, the file will not be written (even if other extensions try to write it, returning false is a hard override). If you return an object or array of objects, it will write a file or multiple files with the details provided. If you leave either of the keys out of the object you return, it will be filled in with the default.

For example, if you want to write the same content, but change where it's written to, you could return an object that has a `path` key specifying a different path.

As another example, 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. Do note that you can also return a promise for your object or array of objects if you need to do async tasks.

You have access to a full `context` object from the write hook, as with any of the other hooks. The context in this hook is an exact mirror of the context that you get in the `after_file` hook.

1.5.11 Example: Concatenation

Now that we have the basic info out of the way, let's take a look at a couple examples of common patterns in roots extensions. First is an example of how to collect the contents of certain files and concatenate them into a single file.

For example, if you were making an extension that collected all the contents of javascript files and concatenated them into a single file, you would want to carefully choose where to store the contents while they were being collected. Let's take a look at some code to put this in context:

```
nodefn = require('when/node/function')
path = require('path')
fs = require('fs')

module.exports (opts) = ->

  class JSConcat
    constructor: ->
```

```
    @category = 'js-concat'
    @contents = ''

  fs: ->
    detect: (f) ->
      path.extname(f.relative) == 'js'

  compile_hooks: ->
    after_file: (ctx) =>
      @contents += ctx.contents
    write: ->
      false

  category_hooks: ->
    after: (ctx) =>
      output = path.join(ctx.roots.config.output_path(), 'build.js')
      nodefn.call(fs.writeFile, output, @contents)
```

What we have here is a simple extension that concatenates js files into a single file rather than outputting them individually. What it does is pretty straightforward. It detects files that have a `.js` extension and puts them into a category. After each one is compiled, it's contents are pushed into a string, and the normal file write is prevented. When they have all been compiled, a file is written to a user-specificed output path containing the concatenated results. The actual write method dives in a little bit, grabbing the roots output path from internal config, and returning a promise, but such is life. Now, this extension works fine, but one small context change would result in a borked extension. See if you can spot the mistake here:

```
# WARNING: This is an example of the *wrong* way to do it.
#          do not actually use this code!

nodefn = require('when/node/function')
path = require('path')
fs = require('fs')

module.exports (opts) = ->
  contents = ''

  class JSConcat
    constructor: ->
      @category = 'js-concat'

    fs: ->
      detect: (f) ->
        path.extname(f.relative) == 'js'

    compile_hooks: ->
      after_file: (ctx) ->
        contents += ctx.contents
      write: ->
        false

    category_hooks: ->
      after: (ctx) =>
        output = path.join(ctx.roots.config.output_path(), 'build.js')
        nodefn.call(fs.writeFile, output, @contents)
```

See it? Can you guess what it would do wrong? What would happen here is that each time a compile happened, it would add another full set of contents to the original. So after three compiles, the file would have 3x the original contents, and they would just be duplicates of the original. This is because while the class is re-instantiated each

compile, the extension itself is only instantiated once when the class is created.