handroll Documentation

Release 4.0

Matt Layman

Jun 18, 2018

Contents

handroll is a static website generator that uses markup languages like Markdown, reStructuredText, and Textile.

handroll development is done on GitHub. Announcements and discussions happen on Google Groups.

CHAPTER 1

Installation

handroll is available for download from PyPI. You can install it with `pip`. handroll is currently supported on Python 3.6, 3.5, 3.4, and PyPy.

```
$ pip install handroll
```

CHAPTER 2

Usage

When inside a website's source directory, the following command will generate results and store them in `output`. Use `handroll -h` to see all the options.

```
$ handroll build
Complete.
```

CHAPTER 3

Features

handroll follows a *"batteries included"* philosophy for generating static websites. One goal is to support a wide range of tools to cater to many diverse interests. The list below isn't exhaustive, but it provides a good idea of what handroll is capable of doing.

- Start a new site with a single command using *Scaffolds* for immediate results.
- Convert Markdown to HTML.
- Convert reStructuredText to HTML.
- Convert Textile to HTML.
- Convert Sass to CSS.
- Copy static assets like CSS or JavaScript.
- Track blog entries and automatically generate a feed (see *Blog extension*).
- Generate a proper Atom XML feed from metadata stored in JSON.
- Produce metadata for popular social media site like Facebook and Twitter.
- Run a development server with the `watch` flag to monitor your site and update the output immediately as you make changes (see *Development server*).
- Find the site source root so you don't have to. If you're anywhere in your site's source, calling `handroll` without the site input parameter will trigger handroll to look for your site's root directory.
- Store global configuration in a configuration file (see *Configuration*). You'll never need to specify the output directory again.
- Keep extra data for templates in a separate front matter section in YAML format (see *Front matter*).
- Templates can use the Jinja2 template engine.
- Content is only updated when either a template or the source file is newer than the existing output file. This eliminates wasted regeneration on unchanged content.
- Be extensible for users who want to write their own plugins (see *Composers* and *Extensions*).
- Provide timing information to see file processing time.

- Translated to many different languages.

The remaining documentation provides additional details about all listed features.

CHAPTER 4

Documentation

4.1 Configuration

handroll supports an optional `handroll.conf` file that can be stored at the root of the site's directory. This `ini` style file provides configuration information that handroll will use while generating the output. For example:

```
[site]
outdir = ~/mblayman.github.io
```

Arguments provided on the command line will override the equivalent configuration file option.

4.1.1 `site` section

The `outdir` option will determine the output directory.

The `outdir` permits relative paths. One useful pattern with relative paths is to set `outdir = ..` as the value. Source and output can exist in a single repository or directory. Putting the output at the root of a repository makes it easy to deploy the entire project as a website. When generating output or watching the source directory, handroll is aware of the source and allows the two directories to coexist without interference.

If a tilde character (~) is supplied, it will be expanded to the user's home directory.

The `with_blog` option set to `true`, `on`, `yes`, or `1` will enable the blog extension. See *Blog extension* for setup information.

4.2 Front matter

Source documents like Markdown files can have additional data added to them. This data is stored in a front matter section at the top of a source document. handroll will read the extra data and pass it along to the template. In the template, the data will be accessible by whatever name was provided. An example Markdown source document would look like:

```
---
title: A Sample Site Title
bonus: It's a secret to everybody.
---
## Another heading

This is text in the body.
```

You may also include the YAML directive (e.g., `%YAML 1.1`). The following example is equally valid.

```
%YAML 1.1
---
title: A Sample Site Title
bonus: It's a secret to everybody.
---
## Another heading

This is text in the body.
```

Note: When using front matter, handroll does not infer the title from the first line of the document. If a title is desired, the attribute must be explicitly added to the front matter.

4.3 Scaffolds

handroll can generate a new site using a single command. This will help you get started in a snap. After making a site with the scaffold command, you can immediately run `handroll` to get a functioning website.

```
$ handroll scaffold default mysite
```

Scaffolds can include any content. The default scaffold includes a base template, an index file in Markdown, a configuration file, and a sample CSS file.

To list the available scaffolds,

```
$ handroll scaffold
```

4.4 Development server

handroll comes with a built-in development server. The server helps develop websites even faster by watching the changes you make. As files in your site are created, modified, or moved, the server will update your output with each change.

The development server is available with the `watch` command. The server will make your site accessible on `http://localhost:8000`.

Here is an example:

```
matt@eden:~/handroll/sample$ handroll watch
Serving /home/matt/handroll/sample/output at http://localhost:8000/.
Press Ctrl-C to quit.
Generating HTML for /home/matt/handroll/sample/index.md ...
```

4.5 Composers

handroll uses a plugin system to decide how to process each file type. The plugins are called composers. A composer is provided a source file and can produce whatever output it desires. handroll will load each available composer using `setuptools` entry points. handroll loads the class and constructs a `Composer` instance by invoking a no parameter constructor.

class `handroll.composers.`**`Composer`**(*config*)

Interface for all composers

`__init__`(*config*)

Each composer is given the configuration when instantiated.

`compose`(*catalog*, *source_file*, *out_dir*)

Compose whatever appropriate output is generated by the composer.

Parameters

- **`catalog`** – the `TemplateCatalog`
- **`source_file`** – the filename of the source
- **`out_dir`** – the directory to store output

`get_output_extension`(*filename*)

Get the extension of the output file generated by this composer.

The filename is required because some composers may vary their extension based on the filename.

`permit_frontmatter`

Check if frontmatter is permitted for the file type.

A plugin should be added to the `handroll.composers` entry point group. For example, the `MarkdownComposer` plugin included by default defines its entry point in `setup.py` as:

```
entry_points={
    'handroll.composers': [
        '.md = handroll.composers.md:MarkdownComposer',
    ]
}
```

This entry point registers the `MarkdownComposer` class in the `handroll.composers.md` module for the `.md` file extension. The example is slightly confusing because the entry point name and the package are the same so here is a fictious example.

A composer class called `FoobarComposer` defined in `another.package` for the `.foobar` file extension would need the following entry point.

```
entry_points={
    'handroll.composers': [
        '.foobar = another.package:FoobarComposer',
    ]
}
```

4.6 Built-in composers

class `handroll.composers.atom.`**`AtomComposer`**(*config*)

Compose an Atom feed from an Atom metadata file (`.atom`).

The AtomComposer parses the metadata specified in the source file and produces an XML Atom feed. AtomComposer uses parameters that are needed by Werkzeug's AtomFeed API. Refer to the Werkzeug documentation for all the available options.

The dates in the feed should be in RfC 3339 format (e.g., 2014-06-13T11:39:30).

Here is a sample feed:

```
{
  "title": "Sample Feed",
  "url": "http://some.website.com/archive.html",
  "id": "http://some.website.com/feed.xml",
  "author": "Matt Layman",
  "entries": [
    {
      "title": "Sample C",
      "updated": "2014-05-04T12:00:00",
      "url": "http://some.website.com/c.html",
      "summary": "A summary of the sample post"
    },
    {
      "title": "Sample B",
      "updated": "2014-03-17T12:00:00",
      "url": "http://some.website.com/b.html",
      "summary": "A summary of the sample post"
    },
    {
      "title": "Sample A",
      "updated": "2014-02-23T00:00:00",
      "url": "http://some.website.com/a.html",
      "summary": "A summary of the sample post"
    }
  ]
}
```

class handroll.composers.**CopyComposer**(*config*)

Copy a source file to the destination.

CopyComposer is the default composer for any unrecognized file type. The source file will be copied to the output directory unless there is a file with an identical name and content already at the destination.

class handroll.composers.sass.**SassComposer**(*path=None*)

Compose CSS files from Sass files (.scss or .sass).

Sass is a CSS preprocessor to help manage CSS files. The Sass website has great documentation to explain how to use it.

Because Sass is not written in the same language as handroll, it must be installed separately before it can be used. Check out the installation options.

4.7 Extensions

In addition to *Composers*, handroll has an extension system to plug in other functionality. Users enable extensions by adding with_* = true to their site section in the configuration file, where * is the name of the extension. For example, the blog extension is named blog, and with_blog = true will enable it.

Extension authors can use the base Extension to create new extensions. Extensions are never directly called, but an extension can connect to one of handroll's *Signals*.

class handroll.extensions.base.**Extension**(*config*)

A base extension which hooks handler methods to handroll's signals.

on_frontmatter_loaded(*source_file*, *frontmatter*)

Handle the `frontmatter_loaded` signal.

Activate this handler by setting `handle_frontmatter_loaded` to `True` in the extension subclass.

Parameters

- **source_file** – Absolute path of the source file
- **frontmatter** – Dictionary of parsed frontmatter

on_post_composition(*director*)

Handle the `post_composition` signal.

Activate this handler by setting `handle_post_composition` to `True` in the extension subclass.

Parameters director – The director instance

on_pre_composition(*director*)

Handle the `pre_composition` signal.

Activate this handler by setting `handle_pre_composition` to `True` in the extension subclass.

Parameters director – The director instance

Extension authors can include new extensions by adding to the `handroll.extensions` entry point. For example, handroll includes the following entry point in `setup.py`:

```
entry_points={
    'handroll.extensions': [
        'blog = handroll.extensions.blog:BlogExtension',
    ]
}
```

4.8 Built-in extensions

4.8.1 Blog extension

The blog extension allows you to automatically generate an atom feed of blog entries. It can also create an entry list for one of your pages.

Enable the blog extension by adding `with_blog = True` to the `site` section of your configuration file.

Atom feed

The extension requires some additional information to create a valid atom feed. Add a `blog` section to your configuration file with the following fields:

- `atom_author` - The author of the blog
- `atom_id` - A unique identifier for the atom feed. One suggestion is to use the URL *of the feed itself*. For example, `http://www.mattlayman.com/feed.xml`.
- `atom_title` - The title of the blog
- `atom_url` - The URL for the feed. For example, `http://www.mattlayman.com/archive.html`.

To create the atom feed, you need to specify an output path using the `atom_output` option. The path provided is relative to the output directory.

```
[blog]
atom_output = feed.xml
```

In this example, the atom feed would be stored in the root of the output directory with a filename of `feed.xml`.

List page

To create a blog list page, add a `list_template` option to your `blog` section. If you include `list_template`, then you must also include `list_output`. `list_output` is a path relative to the output directory.

When the blog extension generates the list page, the context will receive a `blog_list`. The `blog_list` is an HTML fragment of list item tags. There is one list item tag for every post.

Here is a possible sample template.

```
<html>
<body>
  <ul>
    {{ blog_list }}
  </ul>
</body>
</html>
```

And here is some possible output.

```
<html>
<body>
  <ul>
    <li><a href="/another_post.html">Another post</a></li>
    <li><a href="/a_post.html">First post!</a></li>
  </ul>
</body>
</html>
```

For more complex formatting, the actual blog posts are provided in the context as `posts`.

Blog post frontmatter

A source file is marked as a blog post by setting `blog: True` in the front matter. The blog front matter has required and optional fields.

Required fields:

- `title` - The title of the post
- `date` - The published date of the post. The date should be in RfC 3339 format (e.g., `2015-07-15T12:00:00Z`).

Optional fields:

- `summary` - A summary of the post.

When a blog post is rendered, `post` is added to the context. This `post` data contains `previous` and `next` attributes which point to the previous and next chronological posts. This is useful to include links between blog post pages.

4.8.2 Sitemap extension

The sitemap extension generates a sitemap of your site's HTML content. The generated file will be stored in the root of the output directory as `sitemap.txt`.

Enable the sitemap extension by adding `with_sitemap = True` to the `site` section of your configuration file.

4.8.3 Open Graph extension

The Open Graph extension reads blog post frontmatter and adds `open_graph_metadata` to the template context. The data can be added to the `head` section of the HTML output.

Enable the Open Graph extension by adding `with_open_graph = True` to the `site` section of your configuration file.

The extension produces the metadata for an Open Graph `article`. As the `article` type requires an image, additional configuration is required so that an image is always available.

You must include an `open_graph` section with a `default_image`. The `default_image` is a full URL to an image file. This default image will be used whenever an `image` is not specified in a blog post's frontmatter.

```
[open_graph]
default_image = http://www.example.com/images/og.jpg
```

When `image` is provided in the frontmatter, the value produces a URL that is related to the source file. For example, if you have a post in your site at a path of `posts/my-topic.md` and the frontmatter includes `image: butterfly.jpg`, then the included image URL would become `http://www.example.com/posts/butterfly.jpg`.

`image` can also use an absolute path. If the frontmatter sets `image: /images/ladybug.jpg`, the final URL would be `http://www.example.com/images/ladybug.jpg` regardless of where the blog post is in your site structure.

To include the metadata, add it to your template:

```
<html>
 <head>
   {{ open_graph_metadata }}
 </head>
 <body>
 </body>
</html>
```

4.8.4 Twitter extension

The Twitter extension reads blog post frontmatter and adds `twitter_metadata` to the template context. The data can be added to the `head` section of the HTML output.

Enable the Twitter extension by adding `with_twitter = True` to the `site` section of your configuration file.

The extension produces the metadata for an Twitter `summary` card. As the `summary` type requires an image, additional configuration is required so that an image is always available.

You must include an `twitter` section with a `default_image`. The `default_image` is a full URL to an image file. This default image will be used whenever an `image` is not specified in a blog post's frontmatter.

`image` follows the same rules as described for the Open Graph extension.

The Twitter extension also expects a `site_username` to connect with the relevant Twitter account.

```
[twitter]
default_image = http://www.example.com/images/og.jpg
site_username = @mblayman
```

To include the metadata, add it to your template:

```
<html>
 <head>
   {{ twitter_metadata }}
 </head>
 <body>
 </body>
</html>
```

4.9 Signals

handroll fires various signals while running. These signals provide hooks for extensions to execute additional code. The list of signals is provided below.

4.9.1 frontmatter_loaded

`frontmatter_loaded` fires whenever a file contains a front matter section (see *Front matter*). Any handler function that connects to the signal will be called with:

- `source_file` - The absolute path to the file containing front matter.
- `frontmatter` - The front matter dictionary that was loaded.

4.9.2 pre_composition

`pre_composition` fires before processing the entire site. When the watcher is running (see *Development server*), the signal will fire before handling any file or directory change. Any handler function that connects to the signal will be called with:

- `director` - The director instance that processed the site.

4.9.3 post_composition

`post_composition` fires after processing the entire site. When the watcher is running (see *Development server*), the signal will fire after handling any file or directory change. Any handler function that connects to the signal will be called with:

- `director` - The director instance that processed the site.

4.10 Templates

Your source content (e.g., Markdown or reStructuredText) is read and converted into HTML. After content is converted to HTML, it is passed to a template system as a variable called `content`. Each template system can then insert the HTML content into a template.

handroll supports multiple template systems. Templates are stored in a `templates` directory at the root of your site. Alternatively, if you have very simple needs, you can use a `template.html` file at your site's root.

Any template used from the `templates` directory must be specified using front matter (see *Front matter*) or the default `template.html` will be used. This sample Markdown file uses a string template.

```
%YAML 1.1
---
title: With a different template
template: different.html
---
## Another heading

This is using a different string template.
```

4.10.1 String templates

Any template using the `.html` extension (including the default `template.html`) will uses Python's built-in string templates. String templates are limited to the capabilities of the standard library, but they can support basic needs.

4.10.2 Jinja2 templates

Any template using the `.j2` extension will use the Jinja2 template language. handroll works with Jinja's template inheritance system and the majority of Jinja's other features.

4.11 Translation

4.11.1 For Translators

Translation for handroll is done on handroll's Transifex project. All contributions from any language are welcome. The project should be configured to automatically accept requests to join the translation team.

Translators will be added to the AUTHORS list of contributors for contributions of any size (unless you specifically do not want to be listed).

If you have any problems, please feel free to report an issue on Github. If there is a new language that you want to translate for, please file an issue so it can be added (there is a small amount of software setup to make a new language work with handroll).

4.11.2 For Developers

All user interface strings (aside from flag names) should be translated. Strings are wrapped in a conventional way for gettext using the alias _. The gettext method is configured in the `handroll.i18n` module. Any new code would use that method for strings. For example:

```
from handroll.i18n import _

translated = _('This string is translated.')
```

The `handroll.pot` file is the template that the translators use as the basis for translation. handroll uses Babel to generate the `pot` file. A new `pot` file is generated by executing `python setup.py extract_messages`. The updated `pot` file is automatically synced to Transifex after it is pushed to GitHub.

Deployment

Download the `po` files. The `mo` files are automatically generated with the `sdist` command.

```
$ python transifex.py
$ python setup.py sdist
```

4.12 Releases

4.12.1 Version 4.0, In Development

- Add support for Python 3.6.
- Include an `OpenGraphExtension` to add metadata to blog posts.
- Include a `TwitterExtension` to add metadata to blog posts.
- Add `post` context to blog post rendering which includes `post.previous` and `post.next` posts to link between pages.
- Remove support for Python 2.7.
- Drop support for Python 3.3 (EOL).

4.12.2 Version 3.1, Released December 26, 2016

- Processs Jinja 2 templates for any file with a `.j2` extension with the built-in `Jinja2Composer`.
- Add `SitemapExtension` to generate sitemaps.
- Move version information into the `handroll` package so it is available at runtime.
- Perform continuous integration testing on OS X.
- Include `posts` in the blog feed list to permit more complex list rendering.
- Remove support for Python 2.6

4.12.3 Version 3.0, Released March 7, 2016

- Replaced all flag based commands with sub-commands. This change means all interaction now happens through `handroll build`, `handroll watch`, and `handroll scaffold`.

4.12.4 Version 2.1, Released October 18, 2015

- Create a site quickly with the new scaffold command (e.g., `handroll -s default new_site`)
- Use the SmartyPants library to generate better quotation marks for Markdown.
- Composers can be forced to compose with the `--force` flag.
- Translated to Arabic.
- Relax the frontmatter requirement and don’t force the inclusion of the YAML directive (e.g., `%YAML 1.1`).
- Support Python 3.5.

- An output directory can be a relative path.

4.12.5 Version 2.0, Released July 25, 2015

- Added an extension interface for plugin authors to integrate with various events.
- Added a blog extension to automatically generate an Atom XML feed and blog listing page.
- Translated to Greek.

4.12.6 Version 1.5, Released February 24, 2015

- Translated to Dutch.

4.12.7 Version 1.4, Released December 1, 2014

- A development server (accessible from the `watch` flag) will monitor a site and generate new output files as the source is modified.
- Sass support for `.scss` and `.sass` files.
- Add internationalization (i18n).
- Translated to French, German, Italian, Portuguese, and Spanish.
- Skip certain directories that should not be in output (like a Sass cache).
- Moved project to a GitHub organization to separate from a personal account.
- Include documentation in the release.
- Massive unit test improvements (100% coverage).

4.12.8 Version 1.3, Released September 3, 2014

- Update the appropriate output only when a template or content was modified.
- Use Jinja templates or standard Python string templates.
- Provide YAML formatted front matter to add any data to a template.

4.12.9 Version 1.2, Released July 2, 2014

- Add a basic configuration file to specify the output directory.
- A search for the site root is done when no site path is provided.
- Add timing reporting to find slow composers.
- Update Textile version to enable Python 3 support.
- Generate Atom feeds.
- Drop 3.2 support. Too many dependencies do not support it.

4.12.10 Version 1.1, Released June 1, 2014

- Skip undesirable file types (e.g., Vim .swp files).
- Use Markdown code highlighting (via Pygments) and fenced code extensions.
- All input and output is handled as UTF-8 for better character encoding.
- Run against Python versions 2.6 through 3.4 using Travis CI.
- Add a plugin architecture to support composers for any file type.
- Provide HTML docs at Read the Docs.
- Textile support for `.textile` files.
- ReStructuredText support for `.rst` files.
- Support PyPy.

4.12.11 Version 1.0, Released May 4, 2014

- Initial release of `handroll`
- Copy all file types.
- Convert Markdown to HTML.

Index