Nikola Documentation

Release 7.8.8

The Nikola Contributors

Jul 17, 2017

Contents

Those are the docs for the current GitHub master. It might be incompatible with the stable release. The docs for the stable release are available on the Nikola website.

Please note that some examples of Nikola-specific reST syntax might not be visible in this version of Nikola docs.

CHAPTER 1

The Nikola Handbook

Version 7.8.8

Contents

All You Need to Know

After you have Nikola installed:

Create an empty site (with a setup wizard): `nikola init mysite`

You can create a site with demo files in it with `nikola init --demo mysite`

The rest of these commands have to be executed inside the new `mysite` folder.

Create a post: `nikola new_post`

Edit the post: The filename should be in the output of the previous command. You can also use `nikola new_post -e` to open an editor automatically.

Build the site: `nikola build`

Start the test server and open a browser: `nikola serve -b`

That should get you going. If you want to know more, this manual will always be here for you.

DON'T READ THIS MANUAL. IF YOU NEED TO READ IT I FAILED, JUST USE THE THING.

On the other hand, if anything about Nikola is not as obvious as it should be, by all means tell me about it :-)

What's Nikola and what can you do with it?

Nikola is a static website and blog generator. The very short explanation is that it takes some texts you wrote, and uses them to create a folder full of HTML files. If you upload that folder to a server, you will have a rather full-featured website, done with little effort.

Its original goal is to create blogs, but it supports most kind of sites, and can be used as a CMS, as long as what you present to the user is your own content instead of something the user generates.

Nikola can do:

- A blog (example)
- Your company's site
- Your personal site

- A software project's site (example)
- A book's site

Since Nikola-based sites don't run any code on the server, there is no way to process user input in forms.

Nikola can't do:

- Twitter
- Facebook
- An Issue tracker
- Anything with forms, really (except for *comments*!)

Keep in mind that "static" doesn't mean **boring**. You can have animations or whatever fancy CSS3/HTML5 thingie you like. It only means all that HTML is generated already before being uploaded. On the other hand, Nikola sites will tend to be content-heavy. What Nikola is good at is at putting what you write out there.

Getting Help

Get help here!

TL;DR:

- You can file bugs at the issue tracker
- You can discuss Nikola at the nikola-discuss google group
- You can subscribe to the Nikola Blog
- You can follow Nikola on Twitter

Why Static?

Most "modern" websites are *dynamic* in the sense that the contents of the site live in a database, and are converted into presentation-ready HTML only when a user wants to see the page. That's great. However, it presents some minor issues that static site generators try to solve.

In a static site, the whole site, every page, *everything*, is created before the first user even sees it and uploaded to the server as a simple folder full of HTML files (and images, CSS, etc).

So, let's see some reasons for using static sites:

Security Dynamic sites are prone to experience security issues. The solution for that is constant vigilance, keeping the software behind the site updated, and plain old good luck. The stack of software used to provide a static site, like those Nikola generates, is much smaller (Just a web server).

A smaller software stack implies less security risk.

Obsolescence If you create a site using (for example) WordPress, what happens when WordPress releases a new version? You have to update your WordPress. That is not optional, because of security and support issues. If I release a new version of Nikola, and you don't update, *nothing* happens. You can continue to use the version you have now forever, no problems.

Also, in the longer term, the very foundations of dynamic sites shift. Can you still deploy a blog software based on Django 0.96? What happens when your host stops supporting the PHP version you rely on? And so on.

You may say those are long term issues, or that they won't matter for years. Well, I believe things should work forever, or as close to it as we can make them. Nikola's static output and its input files will work as long as

you can install Python 3.3 or newer under Linux, Windows, or OS X and can find a server that sends files over HTTP. That's probably 10 or 15 years at least.

Also, static sites are easily handled by the Internet Archive.

Cost and Performance On dynamic sites, every time a reader wants a page, a whole lot of database queries are made. Then a whole pile of code chews that data, and HTML is produced, which is sent to the user. All that requires CPU and memory.

On a static site, the highly optimized HTTP server reads the file from disk (or, if it's a popular file, from disk cache), and sends it to the user. You could probably serve a bazillion (technical term) page views from a phone using static sites.

Lock-in On server-side blog platforms, sometimes you can't export your own data, or it's in strange formats you can't use in other services. I have switched blogging platforms from Advogato to PyCs to two homebrew systems, to Nikola, and have never lost a file, a URL, or a comment. That's because I have *always* had my own data in a format of my choice.

With Nikola, you own your files, and you can do anything with them.

Components

Nikola provides the following features:

- Blog support, including:
  - Indexes
  - RSS and Atom feeds
  - Tags and categories, with pages and feeds
  - Author pages and feeds (not generated if `ENABLE_AUTHOR_PAGES` is set to `False` or there is only one author)
  - Archives with custom granularity (yearly or monthly)
  - *Comments*
  - Client-side tag clouds (needs manual configuration)
- Static pages (not part of the blog)
- *Math* rendering (via MathJax)
- Custom output paths for generated pages
- Pretty URLs (without `.html`) that don't need web server support
- Easy page template customization
- Internationalization support (my own blog is English and Spanish)
- Sitemap generation (for search engines)
- Custom deployment (if it's a command, you can use it)
- GitHub Pages deployment
- Themes, easy appearance customization
- *Multiple input formats*, including reStructuredText and Markdown
- Easy-to-create image galleries

- Image thumbnail generation
- Support for displaying source code listings
- Custom search
- Asset (CSS/JS) bundling
- gzip compression (for sending via your web server)
- Open Graph, Twitter Cards
- Hyphenation
- Custom *post processing filters* (eg. for minifying files or better typography)

Getting Started

To set Nikola up and create your first site, read the Getting Started Guide.

Creating a Blog Post

Magic Links

You will want to do things like "link from one post to another" or "link to an image gallery", etc. Sure, you can just figure out the URLs for each thing and use that. Or you can use Nikola's special link URLs. Those are done using the syntax `link://kind/name` and a full list of the included ones is here (BTW, I linked to that using `link://slug/path-handlers`)

To create a new post, the easiest way is to run `nikola new_post`. You will be asked for a title for your post, and it will tell you where the post's file is located.

By default, that file will contain also some extra information about your post ("the metadata"). It can be placed in a separate file by using the `-2` option, but it's generally easier to keep it in a single location.

The contents of your post have to be written (by default) in reStructuredText but you can use a lot of different markups using the `-f` option.

Currently, Nikola supports reStructuredText, Markdown, Jupyter Notebooks, HTML as input, can also use Pandoc for conversion, and has support for BBCode, CreoleWiki, txt2tags, Textile and more via plugins — for more details, read the *input format documentation*. You can learn reStructuredText syntax with the reST quickstart.

Please note that Nikola does not support encodings other than UTF-8. Make sure to convert your input files to that encoding to avoid issues. It will prevent bugs, and Nikola will write UTF-8 output anyway.

You can control what markup compiler is used for each file extension with the `COMPILERS` option. The default configuration expects them to be placed in `posts` but that can be changed (see below, the `POSTS` and `PAGES` options)

This is how it works:

```
$ nikola new_post
Creating New Post
-----------------

Title: How to make money
```

```
Scanning posts....done!
INFO: new_post: Your post's text is at: posts/how-to-make-money.rst
```

The content of that file is as follows:

```
.. title: How to make money
.. slug: how-to-make-money
.. date: 2012-09-15 19:52:05 UTC
.. tags:
.. link:
.. description:
.. type: text

Write your post here.
```

You can edit these files with your favorite text editor, and once you are happy with the contents, generate the pages using `nikola build`.

The post page is generated by default using the `post.tmpl` template, which you can use to customize the output. You can also customize paths and the template filename itself — see *How does Nikola decide where posts should go?*

Metadata fields

Nikola supports many metadata fields in posts. All of them are translatable and almost all are optional.

Basic

title Title of the post. (required)

slug Slug of the post. Used as the last component of the page URL. We recommend and default to using a restricted character set (`a-z0-9-_`) because other symbols may cause issues in URLs. (required)

date Date of the post, defaults to now. Multiple date formats are accepted. Adding a timezone is recommended. (required for posts)

tags Comma-separated tags of the post. Some tags have special meaning, including `draft`, `private`, `mathjax`

category Like tags, except each post can have only one, and they usually have more descriptive names.

guid String used as GUID in RSS feeds and as ID in Atom feeds instead of the permalink.

link Link to original source for content. May be displayed by some themes.

description Description of the post. Used in `<meta>` tags for SEO.

type Type of the post. See *Post Types* for details. Whatever you set here (prepended with `post-`) will become a CSS class of the `<article>` element for this post. Defaults to `text` (resulting in a `post-text` class)

Extra

author Author of the post, will be used in the RSS feed and possibly in the post display (theme-dependent)

enclosure Add an enclosure to this post when it's used in RSS. See more information about enclosures

data Path to an external data file (JSON/YAML/TOML dictionary), relative to `conf.py`. Its keys are available for templates as `post.data('key')`.

Translated posts can have different values for this field, and the correct one will be used.

See *The Global Context and Data files* for more details. This is especially useful used in combination with *shortcodes*.

filters See the *Post Processing Filters* section.

hidetitle Set "True" if you do not want to see the **page** title as a heading of the output html file (does not work for posts).

hyphenate Set "True" if you want this document to be hyphenated even if you have hyphenation disabled by default.

nocomments Set to "True" to disable comments. Example:

password The post will be encrypted and invisible until the reader enters the password. Also, the post's source code will not be available.

WARNING: **DO NOT** use for real confidential data. The algorithm used (RC4) is insecure. The implementation may also be easily brute-forced. Please consider using something else if you need *real* encryption!

More information: Issue #1547

pretty_url Set to "False" to disable pretty URL for this page. Example:

previewimage Designate a preview or other representative image path relative to BASE_URL for use with Open Graph for posts. Adds the image when sharing on social media and many other uses.

```
.. previewimage: /images/looks_great_on_facebook.png
```

The image can be of any size and dimension (services will crop and adapt) but should less than 1 MB and be larger than 300x300 (ideally 600x600).

section Section for the post (instead of inferring from output path; requires `POSTS_SECTION_FROM_META` setting)

template Change the template used to render this page/post specific page. That template needs to either be part of the theme, or be placed in a `templates/` folder inside your site.

```
.. template: story.tmpl
```

To add these metadata fields to all new posts by default, you can set the variable `ADDITIONAL_METADATA` in your configuration. For example, you can add the author metadata to all new posts by default, by adding the following to your configuration:

```
ADDITIONAL_METADATA = {
    'author': 'John Doe'
}
```

Metadata formats

Metadata can be in different formats. Current Nikola versions experimentally supports other metadata formats that make it more compatible with other static site generators. The currently supported metadata formats are:

- reST-style comments (`.. name:  value` — default format)
- Two-file format (reST-style, YAML, TOML)
- Jupyter Notebook metadata
- YAML, between `---` (Jekyll, Hugo)
- TOML, between `+++` (Hugo)
- reST docinfo (Pelican)
- Markdown metadata extension (Pelican)

- HTML meta tags (Pelican)

You can add arbitrary meta fields in any format.

When you create new posts, by default the metadata will be created as reST style comments. If you prefer a different format, you can set the `METADATA_FORMAT` to one of these values:

- `"Nikola"`: reST comments, wrapped in a HTML comment if needed (default)
- `"YAML"`: YAML wrapped in "—"
- `"TOML"`: TOML wrapped in "+++"
- `"Pelican"`: Native markdown metadata or reST docinfo fields. Nikola style for other formats.

reST-style comments

The "traditional" and default meta field format is:

```
.. name: value
```

If you are not using reStructuredText, make sure the fields are in a HTML comment in output.

Two-file format

Meta information can also be specified in separate `.meta` files. Those support reST-style metadata, with names and custom fields. They look like the beginning of our reST files:

```
.. title: How to make money
.. slug: how-to-make-money
.. date: 2012-09-15 19:52:05 UTC
```

You can also use YAML or TOML metadata inside those (with the appropriate markers).

Jupyter Notebook metadata

Jupyter posts can store meta information inside `.ipynb` files by using the `nikola` key inside notebook metadata. It can be edited by using *Edit → Edit Notebook Metadata* in Jupyter. Note that values are currently only strings. Sample metadata (Jupyter-specific information omitted):

```
{
    "nikola": {
        "title": "How to make money",
        "slug": "how-to-make-money",
        "date": "2012-09-15 19:52:05 UTC"
    }
}
```

YAML metadata

YAML metadata should be wrapped by a `---` separator (three dashes) and in that case, the usual YAML syntax is used:

```
---
title: How to make money
slug: how-to-make-money
date: 2012-09-15 19:52:05 UTC
---
```

TOML metadata

TOML metadata should be wrapped by a "+++" separator (three plus signs) and in that case, the usual TOML syntax is used:

```
+++
title = "How to make money"
slug =  "how-to-make-money"
date = "2012-09-15 19:52:05 UTC"
+++
```

reST docinfo

Nikola can extract metadata from reStructuredText docinfo fields and the document itself, too:

```
How to make money
=================

:slug: how-to-make-money
:date: 2012-09-15 19:52:05 UTC
```

To do this, you need `USE_REST_DOCINFO_METADATA = True` in your `conf.py`, and Nikola will hide the docinfo fields in the output if you set `HIDE_REST_DOCINFO = True`.

Note that keys are converted to lowercase automatically.

Markdown metadata

Markdown Metadata only works in Markdown files, and requires the `markdown.extensions.meta` extension (see *MARKDOWN_EXTENSIONS*). The exact format is described in the markdown metadata extension docs.

```
title: How to make money
slug: how-to-make-money
date: 2012-09-15 19:52:05 UTC
```

Note that keys are converted to lowercase automatically.

HTML meta tags

For HTML source files, metadata will be extracted from `meta` tags, and the title from the `title` tag. Following Pelican's behaviour, tags can be put in a "tags" meta tag or in a "keywords" meta tag. Example:

```
<html>
    <head>
        <title>My super title</title>
        <meta name="tags" content="thats, awesome" />
        <meta name="date" content="2012-07-09 22:28" />
```

```
        <meta name="modified" content="2012-07-10 20:14" />
        <meta name="category" content="yeah" />
        <meta name="authors" content="Conan Doyle" />
        <meta name="summary" content="Short version for index and feeds" />
    </head>
    <body>
        This is the content of my super blog post.
    </body>
</html>
```

Mapping metadata from other formats

If you import posts from other engines, those may not work with Nikola out of the box due to differing names. However, you can create a mapping to convert meta field names from those formats into what Nikola expects.

For Pelican, use:

```
METADATA_MAPPING = {
    "rest_docinfo": {"summary": "description", "modified": "updated"},
    "markdown_metadata": {"summary": "description", "modified": "updated"}
    "html_metadata": {"summary": "description", "modified": "updated"}
}
```

For Hugo, use:

```
METADATA_MAPPING = {
    "yaml": {"lastmod": "updated"},
    "toml": {"lastmod": "updated"}
}
```

The following source names are supported: `yaml`, `toml`, `rest_docinfo`, `markdown_metadata`.

Multilingual posts

If you are writing a multilingual site, you can also create a per-language post file (for example: `how-to-make-money.es.txt` with the default TRANSLATIONS_PATTERN, see below). This one can replace metadata of the default language, for example:

- The translated title for the post or page
- A translated version of the page name

The pattern used for finding translations is controlled by the TRANSLATIONS_PATTERN variable in your configuration file.

The default is to put the language code before the file extension, so the German translation of `some_file.rst` should be named `some_file.de.rst`. This is because the TRANSLATIONS_PATTERN variable is by default set to:

```
TRANSLATIONS_PATTERN = "{path}.{lang}.{ext}"
```

Note: Considered languages

Nikola will only look for translation of input files for languages specified in the TRANSLATIONS variable.

In case you translate your posts, you might also want to adjust various other settings so that the generated URLs match the translation. You can find most places in *conf.py* by searching for *(translatable)*. For example, you might want to localize */categories/* (search for *TAG_PATH*), */pages/* and */posts/* (search for *POSTS* and *PAGES*, or see the next section), or how to adjust the URLs for subsequent pages for indexes (search for *INDEXES_PRETTY_PAGE_URL*).

Nikola supports multiple languages for a post (we have almost 50 translations!). If you wish to add support for more languages, check out the Transifex page for Nikola

How does Nikola decide where posts should go?

The place where the post will be placed by `new_post` (the first one that matches the given format) and the final post destination (the first one that matches a given file) is based on the `POSTS` and `PAGES` configuration options. The exact mechanism is explained above the config options in the `conf.py` file, and also reproduced below:

```
# POSTS and PAGES contains (wildcard, destination, template) tuples.
#
# The wildcard is used to generate a list of post source files
# (whatever/thing.rst, for example).
#
# That fragment could have an associated metadata file (whatever/thing.meta),
# and optionally translated files (example for Spanish, with code "es"):
#     whatever/thing.es.rst and whatever/thing.es.meta
#
#     This assumes you use the default TRANSLATIONS_PATTERN.
#
# From those files, a set of HTML fragment files will be generated:
# cache/whatever/thing.html (and maybe cache/whatever/thing.html.es)
#
# These files are combined with the template to produce rendered
# pages, which will be placed at
# output/TRANSLATIONS[lang]/destination/pagename.html
#
# where "pagename" is the "slug" specified in the metadata file.
# The page might also be placed in /destination/pagename/index.html
# if PRETTY_URLS are enabled.
#
# The difference between POSTS and PAGES is that POSTS are added
# to feeds, indexes, tag lists and archives and are considered part
# of a blog, while PAGES are just independent HTML pages.
#
# Finally, note that destination can be translated, i.e. you can
# specify a different translation folder per language. Example:
#     PAGES = (
#         ("pages/*.rst", {"en": "pages", "de": "seiten"}, "story.tmpl"),
#         ("pages/*.md", {"en": "pages", "de": "seiten"}, "story.tmpl"),
#     )

POSTS = (
    ("posts/*.rst", "posts", "post.tmpl"),
    ("posts/*.txt", "posts", "post.tmpl"),
    ("posts/*.html", "posts", "post.tmpl"),
)
PAGES = (
    ("pages/*.rst", "pages", "story.tmpl"),
    ("pages/*.txt", "pages", "story.tmpl"),
    ("pages/*.html", "pages", "story.tmpl"),
)
```

Note: POSTS and PAGES are not flat!

Even if the syntax may suggest you can't, you can create any directory structure you want inside `posts/` or `pages/` and it will be reflected in the output. For example, `posts/foo/bar.txt` would produce `output/posts/foo/bar.html`, assuming the slug is also `bar`.

If you have `PRETTY_URLS` enabled, that would be `output/posts/foo/bar/index.html`.

The `new_post` command

`new_post` will use the *first* path in `POSTS` (or `PAGES` if `-p` is supplied) that ends with the extension of your desired markup format (as defined in `COMPILERS` in `conf.py`) as the directory that the new post will be written into. If no such entry can be found, the post won't be created.

The `new_post` command supports some options:

```
$ nikola help new_post
Purpose: create a new blog post or site page
Usage:   nikola new_post [options] [path]

Options:
  -p, --page                Create a page instead of a blog post. (see also: `nikola␣
↪new_page`)
  -t ARG, --title=ARG       Title for the post.
  -a ARG, --author=ARG      Author of the post.
  --tags=ARG                Comma-separated tags for the post.
  -1                        Create the post with embedded metadata (single file␣
↪format)
  -2                        Create the post with separate metadata (two file format)
  -e                        Open the post (and meta file, if any) in $EDITOR after␣
↪creation.
  -f ARG, --format=ARG      Markup format for the post (use --available-formats for␣
↪list)
  -F, --available-formats   List all available input formats
  -s                        Schedule the post based on recurrence rule
  -i ARG, --import=ARG      Import an existing file instead of creating a placeholder
  -d, --date-path           Create post with date path (eg. year/month/day, see NEW_
↪POST_DATE_PATH_FORMAT in config)
```

The optional `path` parameter tells Nikola exactly where to put it instead of guessing from your config. So, if you do `nikola new_post posts/random/foo.txt` you will have a post in that path, with "foo" as its slug. You can also provide a directory name, in which case Nikola will append the file name for you (generated from title).

The `-d, --date-path` option automates creation of `year/month/day` or similar directory structures. It can be enabled on a per-post basis, or you can use it for every post if you set `NEW_POST_DATE_PATH = True` in conf.py.

```
# Use date-based path when creating posts?
# Can be enabled on a per-post basis with `nikola new_post -d`.
# NEW_POST_DATE_PATH = False

# What format to use when creating posts with date paths?
# Default is '%Y/%m/%d', other possibilities include '%Y' or '%Y/%m'.
# NEW_POST_DATE_PATH_FORMAT = '%Y/%m/%d'
```

Teasers

You may not want to show the complete content of your posts either on your index page or in RSS feeds, but to display instead only the beginning of them.

If it's the case, you only need to add a "magical comment" in your post.

In reStructuredText:

```
.. TEASER_END
```

In Markdown (or basically, the resulting HTML of any format):

```
<!-- TEASER_END -->
```

By default all your RSS feeds will be shortened (they'll contain only teasers) whereas your index page will still show complete posts. You can change this behavior with your `conf.py`: `INDEX_TEASERS` defines whether index page should display the whole contents or only teasers. `FEED_TEASERS` works the same way for your Atom and RSS feeds.

By default, teasers will include a "read more" link at the end. If you want to change that text, you can use a custom teaser:

```
.. TEASER_END: click to read the rest of the article
```

You can override the default value for `TEASER_END` in `conf.py` — for example, the following example will work for `.. more`, and will be compatible with both WordPress and Nikola posts:

```
import re
TEASER_REGEXP = re.compile('<!--\s*(more|TEASER_END)(:(.+))?\s*-->', re.IGNORECASE)
```

Or you can completely customize the link using the `READ_MORE_LINK` option.

```
# A HTML fragment with the Read more... link.
# The following tags exist and are replaced for you:
# {link}        A link to the full post page.
# {read_more}   The string "Read more" in the current language.
# {{            A literal { (U+007B LEFT CURLY BRACKET)
# }}            A literal } (U+007D RIGHT CURLY BRACKET)
# READ_MORE_LINK = '<p class="more"><a href="{link}">{read_more}...</a></p>'
```

Drafts

If you add a "draft" tag to a post, then it will not be shown in indexes and feeds. It *will* be compiled, and if you deploy it it *will* be made available, so use with care. If you wish your drafts to be not available in your deployed site, you can set `DEPLOY_DRAFTS = False` in your configuration. This will not work if lazily include `nikola build` in your `DEPLOY_COMMANDS`.

Also if a post has a date in the future, it will not be shown in indexes until you rebuild after that date. This behavior can be disabled by setting `FUTURE_IS_NOW = True` in your configuration, which will make future posts be published immediately. Posts dated in the future are *not* deployed by default (when `FUTURE_IS_NOW = False`). To make future posts available in the deployed site, you can set `DEPLOY_FUTURE = True` in your configuration. Generally, you want FUTURE_IS_NOW and DEPLOY_FUTURE to be the same value.

Private Posts

If you add a "private" tag to a post, then it will not be shown in indexes and feeds. It *will* be compiled, and if you deploy it it *will* be made available, so it will not generate 404s for people who had linked to it.

Queuing Posts

Some blogs tend to have new posts based on a schedule (for example, every Mon, Wed, Fri) but the blog authors don't like to manually schedule their posts. You can schedule your blog posts based on a rule, by specifying a rule in the `SCHEDULE_RULE` in your configuration. You can either post specific blog posts according to this schedule by using the `--schedule` flag on the `new_post` command or post all new posts according to this schedule by setting `SCHEDULE_ALL = True` in your configuration. (Note: This feature requires that the `FUTURE_IS_NOW` setting is set to `False`)

For example, if you would like to schedule your posts to be on every Monday, Wednesday and Friday at 7am, add the following `SCHEDULE_RULE` to your configuration:

```
SCHEDULE_RULE = 'RRULE:FREQ=WEEKLY;BYDAY=MO,WE,FR;BYHOUR=7;BYMINUTE=0;BYSECOND=0'
```

For more details on how to specify a recurrence rule, look at the iCal specification. Or if you are scared of this format, many calendaring applications (eg. Google Calendar) offer iCal exports, so you can copy-paste the repeat rule from a generated iCal (`.ics`) file (which is a human-readable text file).

Say, you get a free Sunday, and want to write a flurry of new posts, or at least posts for the rest of the week, you would run the `new_post` command with the `--schedule` flag, as many times as you want:

```
$ nikola new_post --schedule
# Creates a new post to be posted on Monday, 7am.
$ nikola new_post -s
# Creates a new post to be posted on Wednesday, 7am.
$ nikola new_post -s
# Creates a new post to be posted on Friday, 7am.
.
.
.
```

All these posts get queued up according to your schedule, but note that you will anyway need to build and deploy your site for the posts to appear online. You can have a cron job that does this regularly.

Post Types

Nikola supports specifying post types, just like Tumblr does. Post types affect the look of your posts, by adding a `post-YOURINPUTHERE` CSS class to the post. Each post can have one and exactly one type. Nikola styles the following types in the default themes:

Name(s)	Description	Styling
text	plain text — default value	standard
micro	"small" (short) posts	big serif font

Indexes

All your posts that are not drafts, private or dated in the future, will be shown in indexes.

Settings

Indexes are put in the INDEX_PATH directory, which defaults to an empty string (site root). The "main" index is index.html, and all the further indexes are index-*.html, respectively.

By default, 10 posts are displayed on an index page. This can be changed with INDEX_DISPLAY_POST_COUNT. Indexes can show full posts or just the teasers, as controlled by the INDEX_TEASERS setting (defaults to False).

Titles of the pages can be controlled by using INDEXES_TITLES, INDEXES_PAGES and INDEXES_PAGES_MAIN settings.

Categories and tags use simple lists by default that show only titles and dates; however, you can switch them to full indexes by using CATEGORY_PAGES_ARE_INDEXES and TAG_PAGES_ARE_INDEXES, respectively.

Something similar happens with authors. To use full indexes in authors, set AUTHOR_PAGES_ARE_INDEXES to True.

Static indexes

Nikola uses *static indexes* by default. This means that index-1.html has the oldest posts, and the newest posts past the first 10 are in index-N.html, where N is the highest number. Only the page with the highest number and the main page (index-N.html and index.html) are rebuilt (the others remain unchanged). The page that appears when you click *Older posts* on the index page, index-N.html, might contain **less than 10 posts** if there are not enough posts to fill up all pages.

This can be disabled by setting INDEXES_STATIC to False. In that mode, index-1.html contains all the newest posts past the first 10 and will always contain 10 posts (unless you have less than 20). The last page, index-N.html, contains the oldest posts, and might contain less than 10 posts. This is how many blog engines and CMSes behave. Note that this will lead to rebuilding all index pages, which might be a problem for larger blogs (with a lot of index pages).

Post taxonomy

There are three taxonomy systems in Nikola, or three ways to organize posts. Those are:

- tags
- categories
- sections

Tags and categories are visible on the *Tags and Categories* page, by default available at /categories/. Each tag/category/section has an index page and feeds.

Tags

Tags are the smallest and most basic of the taxonomy items. A post can have multiple tags, specified using the tags metadata entry (comma-separated). You should provide many tags to help your readers, and perhaps search engines, find content on your site.

Please note that tags are case-sensitive and that you cannot have two tags that differ only in case/punctuation (eg. using nikola in one post and Nikola in another will lead to a crash):

```
ERROR: Nikola: You have tags that are too similar: Nikola and nikola
ERROR: Nikola: Tag Nikola is used in: posts/second-post.rst
ERROR: Nikola: Tag nikola is used in: posts/1.rst
```

Nikola uses some tags to mark a post as "special" — those are `draft`, `private`, `mathjax` (for math support).

You can also generate a tag cloud with the tx3_tag_cloud plugin.

Categories

The next unit for organizing your content are categories. A post can have only one category, specified with the `category` meta tag. Those are *deprecated* and replaced by sections. They are displayed alongside tags. You can have categories and tags with the same name (categories' RSS and HTML files are prefixed with `cat_` by default).

Sections

Sections are the newest feature for taxonomy, and are not supported in themes by default. There are two ways to assign a section to a post:

- through the directory structure (first directory is the section name, eg. `/code/my-project/` is in the *code* category) — your POSTS should have those directories as the second element, eg.

```
POSTS = (
    ('posts/code/*.rst', 'code', 'posts'),
)
```

- through the `section` meta field (requires `POSTS_SECTION_FROM_META` setting; recommended especially for existing sites which should not change the directory hierarchy)

Sections are meant to be used to organize different parts of your blog, parts that are about different topics. Unlike tags, which you should have tens (hundreds?) of, you should ideally have less than 10 sections (though it depends on what your blog needs; there is no hard limit).

With sections, you can also use some custom styling — if you install `husl`, you can use `post.section_color()` from within templates to get a distinct color for the section of a post, which you can then use in some inline CSS for the section name.

You can find some examples and more information in the original announcement

Configuring tags, categories and sections

There are multiple configuration variables dedicated to each of the three taxonomies. You can set:

- `TAG_PATH`, `TAGS_INDEX_PATH`, `CATEGORY_PATH`, `CATEGORY_PREFIX` to configure paths used for tags and categories
- `POST_SECTION_NAME`, `POST_SECTION_TITLE`, `TAG_PAGES_TITLES`, `CATEGORY_PAGES_TITLES` to set friendly section names and titles for index pages
- `POST_SECTION_DESCRIPTIONS`, *TAG_PAGES_DESCRIPTIONS*', `CATEGORY_PAGES_DESCRIPTIONS` to set descriptions for each of the items
- `POST_SECTION_COLORS` to customize colors for each section
- `CATEGORY_ALLOW_HIERARCHIES` and `CATEGORY_OUTPUT_FLAT_HIERARCHIES` to allow hierarchical categories
- `TAG_PAGES_ARE_INDEXES` and `CATEGORY_PAGES_ARE_INDEXES` to display full-size indexes instead of simple post lists
- `WRITE_TAG_CLOUDS` to enable/disable generating tag cloud files
- `HIDDEN_TAGS`. `HIDDEN_CATEGORIES` to make some tags/categories invisible in lists

- `POSTS_SECTION_FROM_META` to use `.. section:` in posts instead of inferring paths from paths

What if I don't want a blog?

If you want a static site that does not have any blog-related elements, see our Creating a Site (Not a Blog) with Nikola guide.

Creating a Page

Pages are the same as posts, except that:

- They are not added to the front page
- They don't appear on the RSS feed
- They use the `story.tmpl` template instead of `post.tmpl` by default

The default configuration expects the page's metadata and text files to be on the `pages` folder, but that can be changed (see `PAGES` option above).

You can create the page's files manually or use the `new_post` command with the `-p` option, which will place the files in the folder that has `use_in_feed` set to False.

In some places (including default directories and templates), pages are called *stories* for historic reasons. Both are synonyms for the same thing: pages that are not blog posts.

Supported input formats

Nikola supports multiple input formats. Out of the box, we have compilers available for:

- reStructuredText (default and pre-configured)
- *Markdown*
- *Jupyter Notebook*
- *HTML*
- *PHP*
- anything *Pandoc* supports (including Textile, DocBook, LaTeX, MediaWiki, TWiki, OPML, Emacs Org-Mode, txt2tags, Microsoft Word .docx, EPUB, Haddock markup)

Plus, we have specialized compilers in the Plugins Index for:

- AsciiDoc
- BBCode
- CommonMark
- IRC logs
- Markmin
- MediaWiki (smc.mw)
- Misaka
- ODT

- Emacs Org-Mode
- reST with HTML 5 output
- Textile
- txt2tags
- CreoleWiki
- WordPress posts

Configuring other input formats

In order to use input formats other than reStructuredText, you need some extra setup.

1. Make sure you have the compiler for the input format you want. Some input formats are supported out-of-the-box, but others must be installed from the Plugins repository. You may also need some extra dependencies. You will get helpful errors if you try to build when missing something.
2. You must ensure the compiler and your desired input file extension is included in the `COMPILERS` dict and does not conflict with any other format. This is extremely important for the pandoc compiler.
3. Finally, you must configure the `POSTS` and `PAGES` tuples. Follow the instructions and the format set by pre-existing entries. Make sure to use the same extension as is set in `COMPILERS` and configure the outputs properly.

Markdown

To use Markdown in your posts/pages, make sure `markdown` is in your `COMPILERS` and that at least one of your desired extensions is defined in `POSTS` and `PAGES`.

You can use Python-Markdown extensions by setting the `MARKDOWN_EXTENSIONS` config option:

```
MARKDOWN_EXTENSIONS = ['fenced_code', 'codehilite', 'extra']
```

Nikola comes with some Markdown Extensions built-in and enabled by default, namely a gist directive, a podcast directive, and `~~strikethrough~~` support.

Jupyter Notebook

To use Jupyter Notebooks as posts/pages, make sure `ipynb` is in your `COMPILERS` and that the `.ipynb` extension is defined in `POSTS` and `PAGES`.

The `-f` argument to `new_post` should be used in the `ipynb@KERNEL` format. It defaults to Python in the version used by Nikola if not specified.

Jupyter Notebooks are also supported in stand-alone listings, if Jupyter support is enabled site-wide.

HTML

To use plain HTML in your posts/pages, make sure `html` is in your `COMPILERS` and that the `.html` extension is defined in `POSTS` and `PAGES`.

PHP

There are two ways of using PHP within Nikola:

1. To use PHP in your posts/pages (inside your site, with the theme and everything), make sure `php` is in your `COMPILERS` and that the `.php` extension is defined in `POSTS` and `PAGES`.
2. To use PHP as standalone files (without any modifications), put them in `files/` (or whatever `FILES_FOLDERS` is configured to).

Pandoc

To use Pandoc, you must uncomment the entry in `COMPILERS` and set the extensions list to your desired extensions while also removing them from their original compilers. The input format is inferred from the extension by Pandoc.

Using Pandoc for reStructuredText, Markdown and other input formats that have a standalone Nikola plugin is **not recommended** as it disables plugins and extensions that are usually provided by Nikola.

Shortcodes

This feature is "inspired" (copied wholesale) from Hugo so I will steal part of their docs too.

A shortcode is a simple snippet inside a content file that Nikola will render using a predefined template or custom code from a plugin.

To use them from plugins, please see Extending Nikola

Using a shortcode

In your content files, a shortcode can be called by using this form:

```
{{% raw %}}{{% name parameters %}}{{% /raw %}}
```

Shortcode parameters are space delimited. Parameters with spaces can be quoted (or backslash escaped).

The first word is always the name of the shortcode. Parameters follow the name. Depending upon how the shortcode is defined, the parameters may be named, positional or both. The format for named parameters models that of HTML with the format name="value".

Some shortcodes use or require closing shortcodes. Like HTML, the opening and closing shortcodes match (name only), the closing being prepended with a slash.

Example of a paired shortcode (note that we don't have a highlight shortcode yet ;-):

```
{{% raw %}}{{% highlight python %}} A bunch of code here {{% /highlight %}}{{% /raw %}
↪}
```

Note: Shortcodes and reStructuredText

In reStructuredText shortcodes may fail because docutils turns URL into links and everything breaks. For some shortcodes there are alternative docutils directives (example, you can use the media **directive** instead of the media shortcode.

Also, you can use the shortcode **role**:

```
:sc:`{{% raw %}}{{% shortcode here %}}{{% /raw %}}`
```

That role passes text unaltered, so shortcodes behave correctly.

Built-in shortcodes

doc Will link to a document in the page, see *Doc role for details*. Example:

```
{{% raw %}}Take a look at {{% doc %}}my other post <creating-a-theme>{{% /doc %}}␣
↪about theme creating.{{% /raw %}}
```

post-list Will show a list of posts, see the *Post List directive for details*

media Display media embedded from a URL, for example, this will embed a youtube video:

```
{{% raw %}}{{% media url="https://www.youtube.com/watch?v=Nck6BZga7TQ" %}}{{% /
↪raw %}}
```

chart Create charts via PyGal. This is similar to the *chart directive* except the syntax is adapted to shortcodes. This is an example:

```
{{% raw %}}{{% chart Bar title='Browser usage evolution (in %)' %}}
x_labels='["2002","2003","2004","2005","2006","2007","2008","2009","2010","2011",
↪"2012"]'%}}
'Firefox', [None, None, 0, 16.6, 25, 31]
'Chrome',  [None, None, None, None, None, None]
'IE',      [85.8, 84.6, 84.7, 74.5, 66, 58.6]
'Others',  [14.2, 15.4, 15.3, 8.9, 9, 10.4]
{{% /chart %}}{{% /raw %}}
```

emoji Insert an emoji. For example:

```
{{% raw %}}{{% emoji crying_face %}}{{% /raw %}}
```

This generates a `span` with `emoji` CSS class, so you can style it with a nice font if you want.

gist Show GitHub gists. If you know the gist's ID, this will show it in your site:

{{% raw %}}{{% gist 2395294 %}} {{% /raw %}}

raw Passes the content along, mostly used so I can write this damn section and you can see the shortcodes instead of them being munged into shortcode **output**. I can't show an example because Inception.

Community shortcodes

Shortcodes created by the community are available in the shortcodes repository on GitHub.

Template-based shortcodes

If you put a template in `shortcodes/` called `mycode.tmpl` then Nikola will create a shortcode called `mycode` you can use. Any options you pass to the shortcode will be available as variables for that template. Non-keyword options will be passed in a tuple variable named `_args`.

The post in which the shortcode is being used is available as the `post` variable, so you can access the title as `post.title`, and data loaded via the `data` field in the metadata using `post.data(key)`.

If you use the shortcode as paired, then the contents between the paired tags will be available in the `data` variable. If you want to access the Nikola object, it will be available as `site`. Use with care :-)

See *Extending Nikola* for detailed information.

For example, if your `shortcodes/foo.tmpl` contains this:

```
This uses the bar variable: ${bar}
```

And your post contains this:

```
{{% raw %}}{{% foo bar=bla %}}{{% /raw %}}
```

Then the output file will contain:

```
This uses the bar variable: bla
```

Finally, you can use a template shortcode without a file, by inserting the template in the shortcode itself:

```
{{% raw %}}{{% template %}}{{% /raw %}}
<ul>
% for foo in bar:
<li>${foo}</li>
% endfor
</ul>
{{% raw %}}{{% /template %}}{{% /raw %}}
```

In that case, the template engine used will be your theme's and the arguments you pass, as well as the global context from your `conf.py`, are available to the template you are creating.

You can use anything defined in your confguration's `GLOBAL_CONTEXT` as variables in your shortcode template, with a caveat: Because of an unfortunate implementation detail (a name conflict), `data` is called `global_data` when used in a shortcode.

If you have some template code that you want to appear in both a template and shortcode, you can put the shared code in a separate template and import it in both places. Shortcodes can import any template inside `templates/` and themes, and call any macros defined in those.

For example, if you define a macro `foo(x, y)` in `templates/shared_sc.tmpl`, you can include `shared_foo.tmpl` in `templates/special_post.tmpl` and `shortcodes/foo.tmpl` and then call the `${shared_foo.foo(x, y)}` macro.

The Global Context and Data files

There is a `GLOBAL_CONTEXT` field in your `conf.py` where you can put things you want to make available to your templates.

It will also contain things you put in a `data/` directory within your site. You can use JSON, YAML or TOML files (with the appropriate file extensions: json/js, yaml/yml, toml/tml) that decode to Python dictionaries. For example, if you create `data/foo.json` containing this:

```
{"bar": "baz"}
```

Then your templates can use things like `${data['foo']['bar']}` and it will be replaced by "baz".

Individual posts can also have a data file. Those are specified using the `data` meta field (path relative to `conf.py`, can be different in different post languages). Those are accessible as eg. `${post.data['bar']}` in templates. *Template-based shortcodes* are a good idea in this case.

Data files can be useful for eg. auto-generated sites, where users provide JSON/YAML/TOML files and Nikola generates a large page with data from all data files. (This is especially useful with some automatic rebuild feature, like those documented in *Deployment*)

Data files are also available as `global_data`, to avoid name conflicts in shortcodes. (`global_data` works everywhere.)

Redirections

If you need a page to be available in more than one place, you can define redirections in your `conf.py`:

```
# A list of redirection tuples, [("foo/from.html", "/bar/to.html")].
#
# A HTML file will be created in output/foo/from.html that redirects
# to the "/bar/to.html" URL. notice that the "from" side MUST be a
# relative URL.
#
# If you don't need any of these, just set to []

REDIRECTIONS = [("index.html", "/weblog/index.html")]
```

It's better if you can do these using your web server's configuration, but if you can't, this will work.

Configuration

The configuration file is called `conf.py` and can be used to customize a lot of what Nikola does. Its syntax is python, but if you don't know the language, it still should not be terribly hard to grasp.

The default `conf.py` you get with Nikola should be fairly complete, and is quite commented.

You surely want to edit these options:

```
# Data about this site
BLOG_AUTHOR = "Your Name"  # (translatable)
BLOG_TITLE = "Demo Site"  # (translatable)
SITE_URL = "https://getnikola.com/"
BLOG_EMAIL = "joe@demo.site"
BLOG_DESCRIPTION = "This is a demo site for Nikola."  # (translatable)
```

Some options are marked with a (translatable) comment above or right next to them. For those options, two types of values can be provided:

- a string, which will be used for all languages
- a dict of language-value pairs, to have different values in each language

---

Note: It is possible to load the configuration from another file by specifying `--conf=path/to/other.file` on Nikola's command line. For example, to build your blog using the configuration file `configurations/test.conf.py`, you have to execute `nikola build --conf=configurations/test.conf.py`.

Customizing Your Site

There are lots of things you can do to personalize your website, but let's see the easy ones!

CSS tweaking Using the default configuration, you can create a `assets/css/custom.css` file under `files/` or in your theme and then it will be loaded from the `<head>` blocks of your site pages. Create it and put your CSS code there, for minimal disruption of the provided CSS files.

If you feel tempted to touch other files in assets, you probably will be better off with a *custom theme*.

If you want to use LESS or Sass for your custom CSS, or the theme you use contains LESS or Sass code that you want to override, you will need to install the LESS plugin or SASS plugin create a `less` or `sass` directory in your site root, put your `.less` or `.scss` files there and a targets file containing the list of files you want compiled.

Template tweaking and creating themes If you really want to change the pages radically, you will want to do a *custom theme*.

Navigation Links The `NAVIGATION_LINKS` option lets you define what links go in a sidebar or menu (depending on your theme) so you can link to important pages, or to other sites.

The format is a language-indexed dictionary, where each element is a tuple of tuples which are one of:

1. A (url, text) tuple, describing a link
2. A (((url, text), (url, text), (url, text)), title) tuple, describing a submenu / sublist.

Example:

```
NAVIGATION_LINKS = {
    DEFAULT_LANG: (
        ('/archive.html', 'Archives'),
        ('/categories/index.html', 'Tags'),
        ('/rss.xml', 'RSS'),
        ((('/foo', 'FOO'),
          ('/bar', 'BAR')), 'BAZ'),
    ),
}
```

Note:

1. Support for submenus is theme-dependent. Only one level of submenus is supported.
2. Some themes, including the default Bootstrap 3 theme, may present issues if the menu is too large. (in `bootstrap3`, the navbar can grow too large and cover contents.)
3. If you link to directories, make sure to follow `STRIP_INDEXES`. If it's set to `True`, end your links with a `/`, otherwise end them with `/index.html` — or else they won't be highlighted when active.

The `SEARCH_FORM` option contains the HTML code for a search form based on duckduckgo.com which should always work, but feel free to change it to something else.

Footer `CONTENT_FOOTER` is displayed, small at the bottom of all pages, I use it for the copyright notice. The default shows a text formed using `BLOG_AUTHOR`, `BLOG_EMAIL`, the date and `LICENSE`. Note you need to use `CONTENT_FOOTER_FORMATS` instead of regular str.format or %-formatting, for compatibility with the translatable settings feature.

BODY_END This option lets you define a HTML snippet that will be added at the bottom of body. The main usage is a Google analytics snippet or something similar, but you can really put anything there. Good place for JavaScript.

SOCIAL_BUTTONS_CODE The `SOCIAL_BUTTONS_CODE` option lets you define a HTML snippet that will be added at the bottom of body. It defaults to a snippet for AddThis, but you can really put anything there. See *social_buttons.html* for more details.

Fancy Dates

Nikola can use various styles for presenting dates.

DATE_FORMAT The date format to use if there is no JS or fancy dates are off. Compatible with Python's `strftime()` syntax.

JS_DATE_FORMAT The date format to use if fancy dates are on. Compatible with `moment.js` syntax.

DATE_FANCINESS = 0 Fancy dates are off, and DATE_FORMAT is used.

DATE_FANCINESS = 1 Dates are recalculated in user's timezone. Requires JavaScript.

DATE_FANCINESS = 2 Dates are recalculated as relative time (eg. 2 days ago). Requires JavaScript.

In order to use fancy dates, your theme must support them. The built-in Bootstrap family supports it, but other themes might not by default.

For Mako:

```
<!-- required scripts -- best handled with bundles -->
<script src="/assets/js/moment-with-locales.min.js"></script>
<script src="/assets/js/fancydates.js"></script>

<!-- fancy dates code -->
<script>
moment.locale("${momentjs_locales[lang]}");
fancydates(${date_fanciness}, ${js_date_format});
</script>
<!-- end fancy dates code -->
```

For Jinja2:

```
<!-- required scripts -- best handled with bundles -->
<script src="/assets/js/moment-with-locales.min.js"></script>
<script src="/assets/js/fancydates.js"></script>

<!-- fancy dates code -->
<script>
moment.locale("{{ momentjs_locales[lang] }}");
fancydates({{ date_fanciness }}, {{ js_date_format }});
</script>
<!-- end fancy dates code -->
```

Adding Files

Any files you want to be in `output/` but are not generated by Nikola (for example, `favicon.ico`) just put it in `files/`. Everything there is copied into `output` by the `copy_files` task. Remember that you can't have files that collide with files Nikola generates (it will give an error).

Important

Don't put any files manually in `output/`. Ever. Really. Maybe someday Nikola will just wipe `output/` (when you run `nikola check -f --clean-files`) and then you will be sorry. So, please don't do that.

If you want to copy more than one folder of static files into `output` you can change the FILES_FOLDERS option:

```
# One or more folders containing files to be copied as-is into the output.
# The format is a dictionary of "source" "relative destination".
# Default is:
# FILES_FOLDERS = {'files': '' }
# Which means copy 'files' into 'output'
```

Custom Themes

If you prefer to have a custom appearance for your site, and modifying CSS files and settings (see *Customizing Your Site* for details) is not enough, you can create your own theme. See the *Theming Nikola* and *Creating a Theme* for more details. You can put them in a `themes/` folder and set `THEME` to the directory name. You can also put them in directories listed in the `EXTRA_THEMES_DIRS` configuration variable.

Getting Extra Themes

There are a few themes for Nikola. They are available at the Themes Index. Nikola has a built-in theme download/install mechanism to install those themes — the `theme` command:

```
$ nikola theme -l
Themes:
-------
blogtxt
bootstrap3-gradients



$ nikola theme -i blogtxt
[2013-10-12T16:46:13Z] NOTICE: theme: Downloading:
https://themes.getnikola.com/v6/blogtxt.zip
[2013-10-12T16:46:15Z] NOTICE: theme: Extracting: blogtxt into themes
```

And there you are, you now have themes/blogtxt installed. It's very rudimentary, but it should work in most cases.

If you create a nice theme, please share it! You can do it as a pull request in the GitHub repository.

One other option is to tweak an existing theme using a different color scheme, typography and CSS in general. Nikola provides a `bootswatch_theme` option to create a custom theme by downloading free CSS files from http://bootswatch.com:

```
$ nikola bootswatch_theme -n custom_theme -s flatly -p bootstrap3
[2013-10-12T16:46:58Z] NOTICE: bootswatch_theme: Creating 'custom_theme' theme
from 'flatly' and 'bootstrap3'
[2013-10-12T16:46:58Z] NOTICE: bootswatch_theme: Downloading:
http://bootswatch.com//flatly/bootstrap.min.css
[2013-10-12T16:46:58Z] NOTICE: bootswatch_theme: Downloading:
http://bootswatch.com//flatly/bootstrap.css
[2013-10-12T16:46:59Z] NOTICE: bootswatch_theme: Theme created. Change the THEME␣
↪setting to "custom_theme" to use it.
```

You can even try what different swatches do on an existing site using their handy bootswatchlet

Play with it, there's cool stuff there. This feature was suggested by clodo.

Deployment

If you can specify your deployment procedure as a series of commands, you can put them in the DEPLOY_COMMANDS option, and run them with `nikola deploy`.

You can have multiple deployment presets. If you run `nikola deploy`, the `default` preset is executed. You can also specify the names of presets you want to run (eg. `nikola deploy default`, multiple presets are allowed).

One caveat is that if any command has a % in it, you should double them.

Here is an example, from my own site's deployment script:

```
DEPLOY_COMMANDS = {'default': [
    'rsync -rav --delete output/ ralsina@lateral.netmanagers.com.ar:/srv/www/lateral',
    'rdiff-backup output ~/blog-backup',
    "links -dump 'http://www.twingly.com/ping2?url=lateral.netmanagers.com.ar'",
]}
```

Other interesting ideas are using git as a deployment mechanism (or any other VCS for that matter), using lftp mirror or unison, or Dropbox. Any way you can think of to copy files from one place to another is good enough.

Deploying to GitHub

Nikola provides a separate command `github_deploy` to deploy your site to GitHub Pages. The command builds the site, commits the output to a gh-pages branch and pushes the output to GitHub. Nikola uses the ghp-import command for this.

In order to use this feature, you need to configure a few things first. Make sure you have `nikola` and `git` installed on your PATH.

1. Initialize a Nikola site, if you haven't already.
2. Initialize a git repository in your Nikola source directory by running:

   ```
   git init .
   git remote add origin git@github.com:user/repository.git
   ```

3. Setup branches and remotes in `conf.py`:
   - `GITHUB_DEPLOY_BRANCH` is the branch where Nikola-generated HTML files will be deployed. It should be `gh-pages` for project pages and `master` for user pages (user.github.io).
   - `GITHUB_SOURCE_BRANCH` is the branch where your Nikola site source will be deployed. We recommend and default to `src`.
   - `GITHUB_REMOTE_NAME` is the remote to which changes are pushed.
   - `GITHUB_COMMIT_SOURCE` controls whether or not the source branch is automatically committed to and pushed. We recommend setting it to `True`, unless you are automating builds with Travis CI.
4. Create a `.gitignore` file. We recommend adding at least the following entries:

```
cache
.doit.db
__pycache__
output
```

5. If you set `GITHUB_COMMIT_SOURCE` to False, you must switch to your source branch and commit to it. Otherwise, this is done for you.
6. Run `nikola github_deploy`. This will build the site, commit the output folder to your deploy branch, and push to GitHub. Your website should be up and running within a few minutes.

If you want to use a custom domain, create your `CNAME` file in `files/CNAME` on the source branch. Nikola will copy it to the output directory. To add a custom commit message, use the `-m` option, followed by your message.

Automated rebuilds with Travis CI

If you want automated rebuilds and GitHub Pages deployment, allowing you to blog from anywhere in the world, follow this guide: Automating Nikola rebuilds with Travis CI.

Automated rebuilds with GitLab

GitLab also offers rebuild automation if you want to use Nikola with GitLab Pages. Check out the example Nikola site on GitLab.

Comments

While Nikola creates static sites, there is a minimum level of user interaction you are probably expecting: comments.

Nikola supports several third party comment systems:

- DISQUS
- IntenseDebate
- LiveFyre
- Moot
- Google+
- Facebook
- isso

By default it will use DISQUS, but you can change by setting `COMMENT_SYSTEM` to one of "disqus", "intensedebate", "livefyre", "moot", "googleplus", "facebook" or "isso"

`COMMENT_SYSTEM_ID`

The value of `COMMENT_SYSTEM_ID` depends on what comment system you are using and you can see it in the system's admin interface.

- For DISQUS it's called the **shortname**
- In IntenseDebate it's the **IntenseDebate site acct**

- In LiveFyre it's the **siteId**
- In Moot it's your **username**
- For Google Plus, `COMMENT_SYSTEM_ID` need not be set. WARNING: this will not work correctly in the test server, needs to be deployed to a real server/URL.
- For Facebook, you need to create an app (turn off sandbox mode!) and get an **App ID**
- For isso, it is the URL of isso (must be world-accessible, encoded with Punycode (if using Internationalized Domain Names) and **have a trailing slash**, default `http://localhost:8080/`)

To use comments in a visible site, you should register with the service and then set the `COMMENT_SYSTEM_ID` option.

I recommend 3rd party comments, and specially DISQUS because:

1. It doesn't require any server-side software on your site
2. They offer you a way to export your comments, so you can take them with you if you need to.
3. It's free.
4. It's damn nice.

You can disable comments for a post by adding a "nocomments" metadata field to it:

```
.. nocomments: True
```

DISQUS Support

In some cases, when you run the test site, you won't see the comments. That can be fixed by adding the disqus_developer flag to the templates but it's probably more trouble than it's worth.

Moot Support

Moot doesn't support comment counts on index pages, and it requires adding this to your `conf.py`:

```
BODY_END = """
<script src="//cdn.moot.it/1/moot.min.js"></script>
"""
EXTRA_HEAD_DATA = """
<link rel="stylesheet" type="text/css" href="//cdn.moot.it/1/moot.css">
<meta name="viewport" content="width=device-width">
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
"""
```

Facebook Support

You need jQuery, but not because Facebook wants it (see Issue #639).

Images and Galleries

To create an image gallery, all you have to do is add a folder inside `galleries`, and put images there. Nikola will take care of creating thumbnails, index page, etc.

If you click on images on a gallery, or on images with links in post, you will see a bigger image, thanks to the excellent colorbox. If don't want this behavior, add an `.islink` class to your image or link. (The behavior is caused by `<a class="image-reference">` if you need to use it outside of galleries and reST thumbnails.)

The gallery pages are generated using the `gallery.tmpl` template, and you can customize it there (you could switch to another lightbox instead of colorbox, change its settings, change the layout, etc.).

Images to be used in normal posts can be placed in the `images` folder. These images will be processed and have thumbnails created just as for galleries, but will then be copied directly to the corresponding path in the `output` directory, so you can reference it from whatever page you like, most easily using the `thumbnail` reST extension. If you don't want thumbnails, just use the `files` folder instead.

The `conf.py` options affecting images and gallery pages are these:

```
# One or more folders containing galleries. The format is a dictionary of
# {"source": "relative_destination"}, where galleries are looked for in
# "source/" and the results will be located in
# "OUTPUT_PATH/relative_destination/gallery_name"
# Default is:
GALLERY_FOLDERS = {"galleries": "galleries"}
# More gallery options:
THUMBNAIL_SIZE = 180
MAX_IMAGE_SIZE = 1280
USE_FILENAME_AS_TITLE = True
EXTRA_IMAGE_EXTENSIONS = []

# If set to False, it will sort by filename instead. Defaults to True
GALLERY_SORT_BY_DATE = True

# Folders containing images to be used in normal posts or pages.
# IMAGE_FOLDERS is a dictionary of the form {"source": "destination"},
# where "source" is the folder containing the images to be published, and
# "destination" is the folder under OUTPUT_PATH containing the images copied
# to the site. Thumbnail images will be created there as well.
IMAGE_FOLDERS = {'images': 'images'}

# Images will be scaled down according to IMAGE_THUMBNAIL_SIZE and MAX_IMAGE_SIZE
# options, but will have to be referenced manually to be visible on the site
# (the thumbnail has ``.thumbnail`` added before the file extension by default,
# but a different naming template can be configured with IMAGE_THUMBNAIL_FORMAT).
IMAGE_THUMBNAIL_SIZE = 400
IMAGE_THUMBNAIL_FORMAT = '{name}.thumbnail{ext}'
```

If you add a reST file in `galleries/gallery_name/index.txt` its contents will be converted to HTML and inserted above the images in the gallery page. The format is the same as for posts.

If you add some image filenames in `galleries/gallery_name/exclude.meta`, they will be excluded in the gallery page.

If `USE_FILENAME_AS_TITLE` is True the filename (parsed as a readable string) is used as the photo caption. If the filename starts with a number, it will be stripped. For example `03_an_amazing_sunrise.jpg` will be render as *An amazing sunrise*.

Here is a demo gallery of historic, public domain Nikola Tesla pictures taken from this site.

Embedding Images

Assuming that you have your pictures stored in a folder called `images` (as configured above), you can embed the same in your posts with the following reST directive:

```
.. image:: /images/tesla.jpg
```

Which is equivalent to the following HTML code:

```
<img src="/images/tesla.jpg">
```

Please take note of the leading forward-slash `/` which refers to the root output directory. (Make sure to use this even if you're not deploying to web server root.)

You can also use thumbnails with the `.. thumbnail::` reST directive. For more details, and equivalent HTML code, see *Thumbnails*.

Handling EXIF Data

Your images contain a certain amount of extra data besides the image itself, called the EXIF metadata. It contains information about the camera you used to take the picture, when it was taken, and maybe even the location where it was taken.

This is both useful, because you can use it in some apps to locate all the pictures taken in a certain place, or with a certain camera, but also, since the pictures Nikola publishes are visible to anyone on the Internet, a privacy risk worth considering (Imagine if you post pictures taken at home with GPS info, you are publishing your home address!)

Nikola has some support for managing it, so let's go through a few scenarios to see which one you prefer.

Strip all EXIF data

Do this if you want to be absolutely sure that no sensitive information should ever leak:

```
PRESERVE_EXIF_DATA = False
EXIF_WHITELIST = {}
```

Preserve all EXIF data

Do this if you really don't mind people knowing where pictures were taken, or camera settings:

```
PRESERVE_EXIF_DATA = True
EXIF_WHITELIST = {'*': '*'}
```

Preserve some EXIF data

Do this if you really know what you are doing. EXIF data comes separated in a few IFD blocks. The most common ones are:

0th Information about the image itself

Exif Information about the camera and the image

1st Information about embedded thumbnails (usually nothing)

thumbnail An embedded thumbnail, in JPEG format (usually nothing)

GPS Geolocation information about the image

Interop Not too interesting at this point.

Each IFD in turn contains a number of tags. For example, 0th contains a ImageWidth tag. You can tell Nikola exactly which IFDs to keep, and within each IFD, which tags to keep, using the EXIF_WHITELIST option.

Let's see an example:

```
PRESERVE_EXIF_DATA = True
EXIF_WHITELIST = {
    "0th": ["Orientation", "ImageWidth", "ImageLength"],
    "Interop": "*",
}
```

So, we preserve EXIF data, and the whitelisted IFDs are "0th" and "Interop". That means GPS, for example, will be totally deleted.

Then, for the Interop IFD, we keep everything, and for the 0th IFD we only keep three tags, listed there.

There is a huge number of EXIF tags, described in the standard

Post Processing Filters

You can apply post processing to the files in your site, in order to optimize them or change them in arbitrary ways. For example, you may want to compress all CSS and JS files using yui-compressor.

To do that, you can use the provided helper adding this in your `conf.py`:

```
FILTERS = {
  ".css": ["filters.yui_compressor"],
  ".js": ["filters.yui_compressor"],
}
```

Where `"filters.yui_compressor"` points to a helper function provided by Nikola in the `filters` module. You can replace that with strings describing command lines, or arbitrary python functions.

If there's any specific thing you expect to be generally useful as a filter, contact me and I will add it to the filters library so that more people use it.

The currently available filters are:

Creating your own filters

You can use any program name that works in place as a filter, like `sed -i` and you can use arbitrary Python functions as filters, too.

If your program doesn't run in-place, then you can use Nikola's `runinplace` function (from the `filters` module). For example, this is how the yui_compressor filter is implemented:

```
from nikola.filters import runinplace
def yui_compressor(infile):
    return runinplace(r'yui-compressor --nomunge %1 -o %2', infile)
```

You can turn any function into a filter using `apply_to_text_file` (for text files to be read in UTF-8) and `apply_to_binary_file` (for files to be read in binary mode).

As a silly example, this would make everything uppercase and totally break your website:

```
import string
from nikola.filters import apply_to_text_file
FILTERS = {
  ".html": [apply_to_text_file(string.upper)]
}
```

filters.html_tidy_nowrap Prettify HTML 5 documents with tidy5

filters.html_tidy_wrap Prettify HTML 5 documents wrapped at 80 characters with tidy5

filters.html_tidy_wrap_attr Prettify HTML 5 documents and wrap lines and attributes with tidy5

filters.html_tidy_mini Minify HTML 5 into smaller documents with tidy5

filters.html_tidy_withconfig Run tidy5 with `tidy5.conf` as the config file (supplied by user)

filters.html5lib_minify Minify HTML5 using html5lib_minify

filters.html5lib_xmllike Format using html5lib

filters.typogrify Improve typography using typogrify

filters.typogrify_sans_widont Same as typogrify without the widont filter

filters.minify_lines **THIS FILTER HAS BEEN TURNED INTO A NOOP** and currently does nothing.

filters.normalize_html Pass HTML through LXML to normalize it. For example, it will resolve `&quot;` to actual quotes. Usually not needed.

filters.yui_compressor Compress CSS/JavaScript using YUI compressor

filters.closure_compiler Compile, compress, and optimize JavaScript Google Closure Compiler

filters.optipng Compress PNG files using optipng

filters.jpegoptim Compress JPEG files using jpegoptim

filters.cssminify Minify CSS using http://cssminifier.com/ (requires Internet access)

filters.jsminify Minify JS using http://javascript-minifier.com/ (requires Internet access)

filters.jsonminify Minify JSON files (strip whitespace and use minimal separators).

filters.xmlminify Minify XML files. Suitable for Nikola's sitemaps and Atom feeds.

filters.add_header_permalinks Add links next to every header, Sphinx-style. You will need to add styling for the *headerlink* class, in *custom.css*, for example:

```
/* Header permalinks */
h1:hover .headerlink, h2:hover .headerlink,
h3:hover .headerlink, h4:hover .headerlink,
h5:hover .headerlink, h6:hover .headerlink {
    display: inline;
}

.headerlink {
    display: none;
    color: #ddd;
    margin-left: 0.2em;
    padding: 0 0.2em;
}
```

```
.headerlink:hover {
    opacity: 1;
    background: #ddd;
    color: #000;
    text-decoration: none;
}
```

Additionally, you can provide a custom list of XPath expressions which should be used for finding headers ({hx} is replaced by headers h1 through h6). This is required if you use a custom theme that does not use "e-content entry-content" as a class for post and page contents.

```
# Default value:
HEADER_PERMALINKS_XPATH_LIST = ['*//div[@class="e-content entry-content"]//{hx}']
# Include *every* header (not recommended):
# HEADER_PERMALINKS_XPATH_LIST = ['*//{hx}']
```

filters.deduplicate_ids Prevent duplicated IDs in HTML output. An incrementing counter is added to offending IDs. If used alongside add_header_permalinks, it will fix those links (it must run **after** that filter)

IDs are numbered from the bottom up, which is useful for indexes (updates appear at the top). There are exceptions, which may be configured using DEDUPLICATE_IDS_TOP_CLASSES — if any of those classes appears sin the document, the IDs are rewritten top-down, which is useful for posts/pages (updates appear at the bottom).

Note that in rare cases, permalinks might not always be *permanent* in case of edits.

```
  DEDUPLICATE_IDS_TOP_CLASSES = ('postpage', 'storypage')

You can also use a file blacklist (``HEADER_PERMALINKS_FILE_BLACKLIST``),
useful for some index pages. Paths include the output directory (eg.
``output/index.html``)
```

You can apply filters to specific posts or pages by using the filters metadata field:

```
.. filters: filters.html_tidy_nowrap, "sed s/foo/bar"
```

Optimizing Your Website

One of the main goals of Nikola is to make your site fast and light. So here are a few tips we have found when setting up Nikola with Apache. If you have more, or different ones, or about other web servers, please share!

1. Use a speed testing tool. I used Yahoo's YSlow but you can use any of them, and it's probably a good idea to use more than one.
2. Enable compression in Apache:

```
AddOutputFilterByType DEFLATE text/html text/plain text/xml text/css text/
↪javascript
```

3. If even after you did the previous step the CSS files are not sent compressed:

```
AddType text/css .css
```

4. Optionally you can create static compressed copies and save some CPU on your server with the GZIP_FILES option in Nikola.

5. The webassets Nikola plugin can drastically decrease the number of CSS and JS files your site fetches.
6. Through the filters feature, you can run your files through arbitrary commands, so that images are recompressed, JavaScript is minimized, etc.
7. The USE_CDN option offloads standard JavaScript and CSS files to a CDN so they are not downloaded from your server.

Math

Nikola supports math input via MathJax (by default) or KaTeX. It is activated via the math roles and directives of reStructuredText and the usual LaTeX delimiters for other input formats.

Configuration

Nikola uses MathJax by default. If you want to use KaTeX (faster and prettier, but may not support every feature yet), set `USE_KATEX = True` in `conf.py`.

To use mathematics in a post, you **must** add the `mathjax` tag, no matter which renderer you are using. (Exception: posts that are Jupyter Notebooks are automatically marked as math)

By default, Nikola will accept `\(...\)` for inline math; `\[...\]` and `$$...$$` for display math. If you want to use the old `$...$` syntax as well (which may conflict with running text!), you need to use special config for your renderer:

```
MATHJAX_CONFIG = """
<script type="text/x-mathjax-config">
MathJax.Hub.Config({
    tex2jax: {
        inlineMath: [ ['$','$'], ["\\\\(","\\\\)"] ],
        displayMath: [ ['$$','$$'], ["\\\\[","\\\\]"] ],
        processEscapes: true
    },
    displayAlign: 'center', // Change this to 'left' if you want left-aligned␣
↪equations.
    "HTML-CSS": {
        styles: {'.MathJax_Display': {"margin": 0}}
    }
});
</script>
"""

KATEX_AUTO_RENDER = """
delimiters: [
    {left: "$$", right: "$$", display: true},
    {left: "\\\\[", right: "\\\\]", display: true},
    {left: "$", right: "$", display: false},
    {left: "\\\\(", right: "\\\\)", display: false}
]
"""
```

(Note: the previous paragraph uses invisible characters to prevent rendering TeX for display, so don't copy the examples with three dots to your posts)

Inline usage

Inline mathematics are produced using the reST *math* **role** or the LaTeX backslash-parentheses delimiters:

Euler's formula: $e^{ix} = \cos x + i \sin x$

In reST:

```
Euler's formula: :math:`e^{ix} = \cos x + i\sin x`
```

In HTML and other input formats:

```
Euler's formula: \(e^{ix} = \cos x + i\sin x\)
```

Note that some input formats (including Markdown) require using **double backslashes** in the delimiters (`\\(inline math\\)`). Please check your output first before reporting bugs.

Display usage

Display mathematics are produced using the reST *math* **directive** or the LaTeX backslash-brackets delimiters:

$$\int \frac{dx}{1+ax} = \frac{1}{a}\ln(1+ax)+C$$

In reST:

```
.. math::

   \int \frac{dx}{1+ax}=\frac{1}{a}\ln(1+ax)+C
```

In HTML and other input formats:

```
\[\int \frac{dx}{1+ax}=\frac{1}{a}\ln(1+ax)+C\]
```

Note that some input formats (including Markdown) require using **double backslashes** in the delimiters (`\\[display math\\]`). Please check your output first before reporting bugs.

reStructuredText Extensions

Nikola includes support for a few directives and roles that are not part of docutils, but which we think are handy for website development.

Includes

Nikola supports the standard reStructuredText `include` directive, but with a catch: filenames are relative to **Nikola site root** (directory with `conf.py`) instead of the post location (eg. `posts/` directory)!

Media

This directive lets you embed media from a variety of sites automatically by just passing the URL of the page. For example here are two random videos:

```
.. media:: http://vimeo.com/72425090

.. media:: http://www.youtube.com/watch?v=wyRpAat5oz0
```

It supports Instagram, Flickr, Github gists, Funny or Die, and dozens more, thanks to Micawber

YouTube

To link to a YouTube video, you need the id of the video. For example, if the URL of the video is http://www.youtube.com/watch?v=8N_tupPBtWQ what you need is **8N_tupPBtWQ**

Once you have that, all you need to do is:

```
.. youtube:: 8N_tupPBtWQ
```

Supported options: `height`, `width`, `align` (one of `left`, `center`, `right`) — all are optional. Example:

```
.. youtube:: 8N_tupPBtWQ
   :align: center
```

Vimeo

To link to a Vimeo video, you need the id of the video. For example, if the URL of the video is http://www.vimeo.com/20241459 then the id is **20241459**

Once you have that, all you need to do is:

```
.. vimeo:: 20241459
```

If you have internet connectivity when generating your site, the height and width of the embedded player will be set to the native height and width of the video. You can override this if you wish:

```
.. vimeo:: 20241459
   :height: 240
   :width: 320
```

Supported options: `height`, `width`, `align` (one of `left`, `center`, `right`) — all are optional.

Soundcloud

This directive lets you share music from http://soundcloud.com You first need to get the ID for the piece, which you can find in the "share" link. For example, if the WordPress code starts like this:

```
[soundcloud url="http://api.soundcloud.com/tracks/78131362" .../]
```

The ID is 78131362 and you can embed the audio with this:

```
.. soundcloud:: 78131362
```

You can also embed playlists, via the *soundcloud_playlist* directive which works the same way.

Supported options: `height`, `width`, `align` (one of `left`, `center`, `right`) — all are optional.

Code

The `code` directive has been included in docutils since version 0.9 and now replaces Nikola's `code-block` directive. To ease the transition, two aliases for `code` directive are provided: `code-block` and `sourcecode`:

```
.. code-block:: python
   :number-lines:

   print("Our virtues and our failings are inseparable")
```

Listing

To use this, you have to put your source code files inside `listings` or whatever folders your `LISTINGS_FOLDERS` variable is set to fetch files from. Assuming you have a `foo.py` inside one of these folders:

```
.. listing:: foo.py python
```

Will include the source code from `foo.py`, highlight its syntax in python mode, and also create a `listings/foo.py.html` page (or in another directory, depending on `LISTINGS_FOLDER`) and the listing will have a title linking to it.

The stand-alone `listings/` pages also support Jupyter notebooks, if they are supported site-wide.

Listings support the same options reST includes support (including various options for controlling which parts of the file are included), and also a `linenos` option for Sphinx compatibility.

The `LISTINGS_FOLDER` configuration variable allows to specify a list of folders where to fetch listings from together with subfolder of the `output` folder where the processed listings should be put in. The default is, `LISTINGS_FOLDERS = {'listings': 'listings'}`, which means that all source code files in `listings` will be taken and stored in `output/listings`. Extending `LISTINGS_FOLDERS` to `{'listings': 'listings', 'code': 'formatted-code'}` will additionally process all source code files in `code` and put the results into `output/formatted-code`.

Note: Formerly, `start-at` and `end-at` options were supported; however, they do not work anymore (since v6.1.0) and you should now use `start-after` and `end-before`, respectively. You can also use `start-line` and `end-line`.

Gist

You can easily embed GitHub gists with this directive, like this:

```
.. gist:: 2395294
```

Producing this:

This degrades gracefully if the browser doesn't support JavaScript.

Thumbnails

To include an image placed in the `images` folder (or other folders defined in `IMAGE_FOLDERS`), use the `thumbnail` directive, like this:

```
.. thumbnail:: /images/tesla.jpg
```

The small thumbnail will be placed in the page, and it will be linked to the bigger version of the image when clicked, using colorbox by default. All options supported by the reST image directive are supported (except `target`). If a body element is provided, the thumbnail will mimic the behavior of the figure directive instead:

```
.. thumbnail:: /images/tesla.jpg

   Nikola Tesla, the man that invented the 20th century.
```

If you want to include a thumbnail in a non-reST post, you need to produce at least this basic HTML:

```
<a class="image-reference" href="images/tesla.jpg"><img src="images/tesla.thumbnail.
↪jpg"></a>
```

Chart

This directive is a thin wrapper around Pygal and will produce charts as SVG files embedded directly in your pages.

Here's an example of how it works:

```
.. chart:: Bar
   :title: 'Browser usage evolution (in %)'
   :x_labels: ["2002", "2003", "2004", "2005", "2006", "2007"]

   'Firefox', [None, None, 0, 16.6, 25, 31]
   'Chrome',  [None, None, None, None, None, None]
   'IE',      [85.8, 84.6, 84.7, 74.5, 66, 58.6]
   'Others',  [14.2, 15.4, 15.3, 8.9, 9, 10.4]
```

The argument passed next to the directive (Bar in that example) is the type of chart, and can be one of Line, StackedLine, Bar, StackedBar, HorizontalBar, XY, DateY, Pie, Radar, Dot, Funnel, Gauge, Pyramid. For examples of what each kind of graph is, check here

It can take *a lot* of options to let you customize the charts (in the example, title and x_labels). You can use any option described in the pygal docs

Finally, the content of the directive is the actual data, in the form of a label and a list of values, one series per line.

Doc

This role is useful to make links to other post or page inside the same site.

Here's an example:

```
Take a look at :doc:`my other post <creating-a-theme>` about theme creating.
```

In this case we are giving the portion of text we want to link. So, the result will be:

> Take a look at *my other post* about theme creating.

If we want to use the post's title as the link's text, just do:

```
Take a look at :doc:`creating-a-theme` to know how to do it.
```

and it will produce:

Take a look at *Creating a Theme* to know how to do it.

Post List

Warning: Any post or page that uses this directive will be considered out of date, every time a post is added or deleted, causing maybe unnecessary rebuilds.

On the other hand, it will sometimes **not** be considered out of date if a post content changes, so it can sometimes be shown outdated, in those cases, use `nikola build -a` to force a total rebuild.

This directive can be used to generate a list of posts. You could use it, for example, to make a list of the latest 5 blog posts, or a list of all blog posts with the tag `nikola`:

```
Here are my 5 latest and greatest blog posts:

.. post-list::
   :stop: 5

These are all my posts about Nikola:

.. post-list::
   :tags: nikola
```

Using shortcode syntax (for other compilers):

```
{{% raw %}}{{% post-list stop=5 %}}{{% /post-list %}}{{% /raw %}}
```

The following options are recognized:

- **`start`** [integer] The index of the first post to show. A negative value like `-3` will show the *last* three posts in the post-list. Defaults to None.
- **`stop`** [integer] The index of the last post to show. A value negative value like `-1` will show every post, but not the *last* in the post-list. Defaults to None.
- **`reverse`** [flag] Reverse the order of the post-list. Defaults is to not reverse the order of posts.
- **`sort`: string** Sort post list by one of each post's attributes, usually `title` or a custom `priority`. Defaults to None (chronological sorting).
- **`date`: string** Show posts that match date range specified by this option. Format:
  - comma-separated clauses (AND)
  - **clause: attribute comparison_operator value (spaces optional)**
    - attribute: year, month, day, hour, month, second, weekday, isoweekday; or empty for full datetime
    - comparison_operator: == != <= >= < >
    - value: integer, 'now' or dateutil-compatible date input
- **`tags`** [string [, string...]] Filter posts to show only posts having at least one of the `tags`. Defaults to None.
- **`require_all_tags`** [flag] Change tag filter behaviour to show only posts that have all specified `tags`. Defaults to False.
- **`categories`** [string [, string...]] Filter posts to show only posts having one of the `categories`. Defaults to None.

- **sections** [string [, string...]] Filter posts to show only posts having one of the `sections`. Defaults to None.
- **slugs** [string [, string...]] Filter posts to show only posts having at least one of the `slugs`. Defaults to None.
- **post_type (or type)** [string] Show only `posts`, `pages` or `all`. Replaces `all`. Defaults to `posts`.
- **all** [flag] (deprecated, use `post_type` instead) Shows all posts and pages in the post list. Defaults to show only posts.
- **lang** [string] The language of post *titles* and *links*. Defaults to default language.
- **template** [string] The name of an alternative template to render the post-list. Defaults to `post_list_directive.tmpl`
- **id** [string] A manual id for the post list. Defaults to a random name composed by `'post_list_' + uuid.uuid4().hex`.

The post list directive uses the `post_list_directive.tmpl` template file (or another one, if you use the `template` option) to generate the list's HTML. By default, this is an unordered list with dates and clickable post titles. See the template file in Nikola's base theme for an example of how this works.

The list may fail to update in some cases, please run `nikola build -a` with the appropriate path if this happens.

We recommend using pages with dates in the past (1970-01-01) to avoid dependency issues.

If you are using this as a shortcode, flags (`reverse`, `all`) are meant to be used with a `True` argument, eg. `all=True`.

Importing your WordPress site into Nikola

If you like Nikola, and want to start using it, but you have a WordPress blog, Nikola supports importing it. Here are the steps to do it:

1. Get an XML dump of your site[1]
2. `nikola import_wordpress mysite.wordpress.2012-12-20.xml`

After some time, this will create a `new_site` folder with all your data. It currently supports the following:

- All your posts and pages
- Keeps "draft" status
- Your tags and categories
- Imports your attachments and fixes links to point to the right places
- Will try to add redirects that send the old post URLs to the new ones
- Will give you a URL map so you know where each old post was

  This is also useful for DISQUS thread migration, or server-based 301 redirects!
- Allows you to export your comments with each post
- Exports information on attachments per post
- There are different methods to transfer the content of your posts:

[1] The dump needs to be in 1.2 format. You can check by reading it, it should say `xmlns:excerpt="http://wordpress.org/export/1.2/excerpt/"` near the top of the file. If it says `1.1` instead of `1.2` you will have to update your WordPress before dumping.
Other versions may or may not work.

- You can convert them to HTML with the WordPress page compiler plugin for Nikola. This will format the posts including supported shortcodes the same way as WordPress does. Use the `--transform-to-html` option to convert your posts to HTML.

  If you use this option, you do not need to install the plugin permanently. You can ask Nikola to install the plugin into the subdirectory `plugins` of the current working directory by specifying the `--install-wordpress-compiler` option.

- You can leave the posts the way they are and use the WordPress page compiler plugin to render them when building your new blog. This also allows you to create new posts using the WordPress syntax, or to manually add more shortcode plugins later. Use the `--use-wordpress-compiler` option to not touch your posts.

  If you want to use this option, you have to install the plugin permanently. You can ask Nikola to install the plugin into your new site by specifying the `--install-wordpress-compiler` option.

- You can let Nikola convert your posts to Markdown. This is *not* error free, because WordPress uses some unholy mix of HTML and strange things. This is the default option and requires no plugins.

You will find your old posts in `new_site/posts/post-title.html` in the first case, `new_site/posts/post-title.wp` in the second case or `new_site/posts/post-title.md` in the last case if you need to edit or fix any of them.

Please note that the page compiler currently only supports the `[code]` shortcode, but other shortcodes can be supported via plugins.

Also note that the WordPress page compiler is licensed under GPL v2 since it uses code from WordPress itself, while Nikola is licensed under the more liberal MIT license.

This feature is a work in progress, and the only way to improve it is to have it used for as many sites as possible and make it work better each time, so we are happy to get requests about it.

Importing to a custom location or into an existing site

It is possible to either import into a location you desire or into an already existing Nikola site. To do so you can specify a location after the dump:

```
$ nikola import_wordpress mysite.wordpress.2012-12-20.xml -o import_location
```

With this command Nikola will import into the folder `import_location`.

If the folder already exists Nikola will not overwrite an existing `conf.py`. Instead a new file with a timestamp at the end of the filename will be created.

Using Twitter Cards

Nikola supports Twitter Card summaries, but they are disabled by default.

Twitter Cards enable you to show additional information in Tweets that link to your content. Nikola supports Twitter Cards. They are implemented to use *Open Graph* tags whenever possible.

Important

To use Twitter Cards you need to opt-in on Twitter. To do so, please visit https://cards-dev.twitter.com/validator

---

Images displayed come from the *previewimage* meta tag.

You can specify the card type by using the *card* parameter in TWITTER_CARD.

To enable and configure your use of Twitter Cards, please modify the corresponding lines in your `conf.py`:

```
TWITTER_CARD = {
    'use_twitter_cards': True,  # enable Twitter Cards
    'card': 'summary',          # Card type, you can also use 'summary_large_image',
                                # see https://dev.twitter.com/cards/types
    'site': '@website',         # twitter nick for the website
    'creator': '@username',     # Username for the content creator / author.
}
```

Custom Plugins

You can create your own plugins (see *Extending Nikola*) and use them in your own site by putting them in a `plugins/` folder. You can also put them in directories listed in the `EXTRA_PLUGINS_DIRS` configuration variable.

Getting Extra Plugins

If you want extra plugins, there is also the Plugins Index.

Similarly to themes, there is a nice, built-in command to manage them — `plugin`:

```
$ nikola plugin -l
Plugins:
--------
helloworld
tags



$ nikola plugin --install helloworld
[2013-10-12T16:51:56Z] NOTICE: install_plugin: Downloading: https://plugins.getnikola.
↪com/v6/helloworld.zip
[2013-10-12T16:51:58Z] NOTICE: install_plugin: Extracting: helloworld into plugins
plugins/helloworld/requirements.txt
[2013-10-12T16:51:58Z] NOTICE: install_plugin: This plugin has Python dependencies.
[2013-10-12T16:51:58Z] NOTICE: install_plugin: Installing dependencies with pip...


[2013-10-12T16:51:59Z] NOTICE: install_plugin: Dependency installation succeeded.
[2013-10-12T16:51:59Z] NOTICE: install_plugin: This plugin has a sample config file.
Contents of the conf.py.sample file:

    # Should the Hello World plugin say "BYE" instead?
    BYE_WORLD = False
```

Then you also can uninstall your plugins:

```
$ nikola plugin --uninstall tags
[2014-04-15T08:59:24Z] WARNING: plugin: About to uninstall plugin: tags
[2014-04-15T08:59:24Z] WARNING: plugin: This will delete /home/ralsina/foo/plugins/
↪tags
Are you sure? [y/n] y
[2014-04-15T08:59:26Z] WARNING: plugin: Removing /home/ralsina/foo/plugins/tags
```

And upgrade them:

```
$ nikola plugin --upgrade
[2014-04-15T09:00:18Z] WARNING: plugin: This is not very smart, it just reinstalls␣
↪some plugins and hopes for the best
Will upgrade 1 plugins: graphviz
Upgrading graphviz
[2014-04-15T09:00:20Z] INFO: plugin: Downloading: https://plugins.getnikola.com/v7/
↪graphviz.zip
[2014-04-15T09:00:20Z] INFO: plugin: Extracting: graphviz into /home/ralsina/.nikola/
↪plugins/
[2014-04-15T09:00:20Z] NOTICE: plugin: This plugin has third-party dependencies you␣
↪need to install manually.
Contents of the requirements-nonpy.txt file:

    Graphviz
        http://www.graphviz.org/

You have to install those yourself or through a package manager.
```

You can also share plugins you created with the community! Visit the GitHub repository to find out more.

You can use the plugins in this repository without installing them into your site, by cloning the repository and adding the path of the plugins directory to the `EXTRA_PLUGINS_DIRS` list in your configuration.

Advanced Features

Debugging

For pdb debugging in Nikola, you should use `doit.tools.set_trace()` instead of the usual pdb call. By default, doit (and thus Nikola) redirects stdout and stderr. Thus, you must use the different call. (Alternatively, you could run with `nikola build -v 2`, which disables the redirections.)

To show more logging messages, as well as full tracebacks, you need to set an environment variable: `NIKOLA_DEBUG=1`. If you want to only see tracebacks, set `NIKOLA_SHOW_TRACEBACKS=1`.

Shell Tab Completion

Since Nikola is a command line tool, and this is the 21st century, it's handy to have smart tab-completion so that you don't have to type the full commands.

To enable this, you can use the `nikola tabcompletion` command like this, depending on your shell:

```
$ nikola tabcompletion --shell bash --hardcode-tasks > _nikola_bash
$ nikola tabcompletion --shell zsh --hardcode-tasks > _nikola_zsh
```

The `--hardcode-tasks` adds tasks to the completion and may need updating periodically.

Please refer to your shell's documentation for help on how to use those files.

License

Nikola is released under the MIT license, which is a free software license. Some components shipped along with Nikola, or required by it are released under other licenses.

If you are not familiar with free software licensing, here is a brief explanation (this is NOT legal advice): In general, you can do pretty much anything you want — including modifying Nikola, using and redistributing the original version or the your modified version. However, if you redistribute Nikola to someone else, either a modified version or the original version, the full copyright notice and license text must be included in your distribution. Nikola is provided "as is", and the Nikola contributors are not liable for any damage caused by the software. Read the full license text for details.

CHAPTER 2

Creating a Site (Not a Blog) with Nikola

One of the most frequent questions I get about Nikola is "but how do I create a site that's not a blog?". And of course, that's because the documentation is heavily blog–oriented. This document will change that ;-)

Since it started, Nikola has had the capabilities to create generic sites. For example, Nikola's own site is a fairly generic one. Let's go step by step on how you can do something like that.

As usual when starting a nikola site, you start with `nikola init` which creates a empty (mostly) configured site:

```
$ nikola init mysite
Creating Nikola Site
====================

[1970-01-01T00:00:00Z] INFO: init: Created empty site at mysite.
```

Then we go into the new `mysite` folder, and make the needed changes in the `conf.py` configuration file:

```
# Data about this site
BLOG_AUTHOR = "Roberto Alsina"
BLOG_TITLE = "Not a Blog"
# This is the main URL for your site. It will be used
# in a prominent link
SITE_URL = "https://getnikola.com/"
BLOG_EMAIL = "ralsina@example.com"
BLOG_DESCRIPTION = "This is a demo site (not a blog) for Nikola."

#
# Some things in the middle you don't really need to change...
#

# you can also keep the current content of POSTS if you want a blog with your site
POSTS = ()
# remove destination directory to generate pages in the root directory
PAGES = (
    ("pages/*.rst", "", "story.tmpl"),
    ("pages/*.txt", "", "story.tmpl"),
```

```
    ("pages/*.html", "", "story.tmpl"),
)

# And to avoid a conflict because blogs try to generate /index.html
INDEX_PATH = "blog"

# Or you can disable blog indexes altogether:
# DISABLE_INDEXES_PLUGIN_INDEX_AND_ATOM_FEED = True
```

And now we are ready to create our first page:

```
$ nikola new_page
Creating New Page
-----------------

Title: index
Scanning posts....done!
[1970-01-01T00:00:00Z] INFO: new_page: Your page's text is at: pages/index.rst
```

We can now build and preview our site:

```
$ nikola build
Scanning posts.done!
.  render_site:output/categories/index.html
.  render_sources:output/index.txt
.  render_rss:output/rss.xml

$ nikola serve
[1970-01-01T00:00:00Z] INFO: serve: Serving HTTP on 0.0.0.0 port 8000...
```

And you can see your (very empty) site in http://localhost:8000/

So, what's in that `pages/index.txt` file?

```
.. title: index
.. slug: index
.. date: 1970-01-01 00:00:00 UTC
.. tags:
.. link:
.. description:


Write your post here.
```

`title` is the page title, `slug` is the name of the generated HTML file (in this case it would be `index.html`). `date`, `tags` and `link` doesn't matter at all in pages. `description` is useful for SEO purposes if you care for that.

And below, the content. By default Nikola uses reStructuredText but it supports a ton of formats, including Markdown, plain HTML, Jupyter Notebooks, BBCode, Wiki, and Textile. We will use reStructuredText for this example, but some people might find it a bit too limiting — if that is the case, try using HTML for your pages (Nikola does this on the index page, for example).

So, let's give the page a nicer title, and some fake content. Since the default Nikola theme (called `bootstrap3`) is based on Bootstrap you can use anything you like from it:

```
.. title: Welcome To The Fake Site
.. slug: index
.. date: 1970-01-01 00:00:00 UTC
```

```
.. tags:
.. link:
.. description: Fake Site version 1, welcome page!


.. class:: jumbotron col-md-6

.. admonition:: This is a Fake Site

    It pretends to be about things, but is really just an example.

    .. raw:: html

       <a href="https://getnikola.com/" class="btn btn-primary btn-lg">Click Me!</a>


.. class:: col-md-5

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris non nunc turpis.
Phasellus a ullamcorper leo. Sed fringilla dapibus orci eu ornare. Quisque
gravida quam a mi dignissim consequat. Morbi sed iaculis mi. Vivamus ultrices
mattis euismod. Mauris aliquet magna eget mauris volutpat a egestas leo rhoncus.
In hac habitasse platea dictumst. Ut sed mi arcu. Nullam id massa eu orci
convallis accumsan. Nunc faucibus sodales justo ac ornare. In eu congue eros.
Pellentesque iaculis risus urna. Proin est lorem, scelerisque non elementum at,
semper vel velit. Phasellus consectetur orci vel tortor tempus imperdiet. Class
aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos
himenaeos.
```

TIP: Nice URLs

If you like your URLs without the `.html` then you want to create folders and put the pages in `index.html` inside them using the `PRETTY_URLS` option (on by default)

And that's it. You will want to change the NAVIGATION_LINKS option to create a reasonable menu for your site, you may want to modify the theme (check `nikola help bootswatch_theme` for a quick & dirty solution), and you may want to add a blog later on, for company news or whatever.

TIP: So, how do I add a blog now?

First, change the `POSTS` option like this:

```
POSTS = (
    ("posts/*.rst", "blog", "post.tmpl"),
    ("posts/*.txt", "blog", "post.tmpl"),
    ("posts/*.html", "blog", "post.tmpl"),
)
```

Create a post with `nikola new_post` and that's it, you now have a blog in the `/blog/` subdirectory of your site — you may want to link to it in `NAVIGATION_LINKS`.

If you want to see a site implementing all of the above, check out the Nikola website.

I hope this was helpful!

CHAPTER 3

Creating a Theme

Nikola is a static site and blog generator. So is Jekyll. While I like what we have done with Nikola, I do admit that Jekyll (and others!) have many more, and nicer themes than Nikola does.

This document is an attempt at making it easier for 3rd parties (that means *you* people! ;-) to create themes. Since I **suck** at designing websites, I asked for opinions on themes to port, and got some feedback. Since this is **Not So Hard™**, I will try to make time to port a few and see what happens.

If you are looking for a reference, check out *Theming reference* and Template variables.

Today's theme is Lanyon which is written by @mdo and released under a MIT license, which is liberal enough.

So, let's get started.

Checking It Out

The first step in porting a theme is making the original theme work. Lanyon is awesome in that its GitHub project is a full site!

So:

```
# Get jekyll
sudo apt-get install jekyll

# Get Lanyon
git clone git@github.com:poole/lanyon.git

# Build it
cd lanyon && jekyll build

# Look at it
jekyll serve & google-chrome http://localhost:4000
```

If you **do not want to install Jekyll**, you can also see it in action at http://lanyon.getpoole.com/

Some things jump to my mind:

1. This is one fine looking theme
2. Very clear and readable
3. Nice hidden navigation-thingy

Also, from looking at the project's README it supports some nice configuration options:

1. Color schemes
2. Reverse layout
3. Sidebar overlay instead of push
4. Open the sidebar by default, or on a per-page basis by using its metadata

Let's try to make all those nice things survive the porting.

Starting From Somewhere

Nikola has a nice, clean, base theme from which you can start when writing your own theme. Why start from that instead of from a clean slate? Because theme inheritance is going to save you a ton of work, that's why. If you start from scratch you won't be able to build **anything** until you have a bunch of templates written. Starting from base, you just need to hack on the things you **need** to change.

First, we create a site with some content in it. We'll use the `nikola init` wizard (with the `--demo` option) for that:

```
$ nikola init --demo lanyon-port
Creating Nikola Site
====================

This is Nikola v7.8.0.  We will now ask you a few easy questions about your new site.
If you do not want to answer and want to go with the defaults instead, simply restart␣
↪with the `-q` parameter.
--- Questions about the site ---
Site title [My Nikola Site]:
Site author [Nikola Tesla]:
Site author's e-mail [n.tesla@example.com]:
Site description [This is a demo site for Nikola.]:
Site URL [https://example.com/]:
--- Questions about languages and locales ---
We will now ask you to provide the list of languages you want to use.
Please list all the desired languages, comma-separated, using ISO 639-1 codes.  The␣
↪first language will be used as the default.
Type '?' (a question mark, sans quotes) to list available languages.
Language(s) to use [en]:

Please choose the correct time zone for your blog. Nikola uses the tz database.
You can find your time zone here:
http://en.wikipedia.org/wiki/List_of_tz_database_time_zones

Time zone [UTC]:
    Current time in UTC: 16:02:07
Use this time zone? [Y/n]
--- Questions about comments ---
You can configure comments now.  Type '?' (a question mark, sans quotes) to list␣
↪available comment systems.  If you do not want any comments, just leave the field␣
↪blank.
```

```
Comment system:

That's it, Nikola is now configured.  Make sure to edit conf.py to your liking.
If you are looking for themes and addons, check out https://themes.getnikola.com/ and␣
↪https://plugins.getnikola.com/.
Have fun!
[2015-05-28T16:02:08Z] INFO: init: A new site with example data has been created at␣
↪lanyon-port.
[2015-05-28T16:02:08Z] INFO: init: See README.txt in that folder for more information.
```

Then, we create an empty theme inheriting from base. This theme will use Mako templates. If you prefer Jinja2, then you should use `base-jinja` as a parent and `jinja` as engine instead:

```
$ cd lanyon-port/
$ nikola theme -n lanyon --parent base --engine mako
```

Edit `conf.py` and set `THEME = 'lanyon'`. Also set `USE_BUNDLES = False` (just do it for now, we'll get to bundles later). Also, if you intend to publish your theme on the Index, or want to use it with older versions (v7.8.5 or older), use the `--legacy-meta` option for `nikola theme -n`.

You can now build that site using `nikola build` and it will look like this:

Fig. 3.1: This is just the base theme.

Basic CSS

The next step is to know exactly how Lanyon's pages work. To do this, we read its HTML. First let's look at the head element:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en-us">

<head>
<link href="http://gmpg.org/xfn/11" rel="profile">
<meta http-equiv="content-type" content="text/html; charset=utf-8">

<!-- Enable responsiveness on mobile devices-->
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1
↪">

<title>
    Lanyon &middot; A Jekyll theme
</title>

<!-- CSS -->
<link rel="stylesheet" href="/public/css/poole.css">
<link rel="stylesheet" href="/public/css/syntax.css">
<link rel="stylesheet" href="/public/css/lanyon.css">
<link rel="stylesheet" href="http://fonts.googleapis.com/css?family=PT+Serif:400,
↪400italic,700|PT+Sans:400">

<!-- Icons -->
<link rel="apple-touch-icon-precomposed" sizes="144x144" href="/public/apple-touch-
↪icon-144-precomposed.thumbnail.png">
```

```
<link rel="shortcut icon" href="/public/favicon.ico">

<!-- RSS -->
<link rel="alternate" type="application/rss+xml" title="RSS" href="/atom.xml">

<!-- Google Analytics -->
[...]
</head>
```

The interesting part there is that it loads a few CSS files. If you check the source of your Nikola site, you will see something fairly similar:

```
<!DOCTYPE html>
<html prefix="og: http://ogp.me/ns# article: http://ogp.me/ns/article# " vocab="http:/
↪/ogp.me/ns" lang="en">
<head>
<meta charset="utf-8">
<meta name="description" content="This is a demo site for Nikola.">
<meta name="viewport" content="width=device-width">
<title>My Nikola Site | My Nikola Site</title>

<link href="assets/css/html4css1.css" rel="stylesheet" type="text/css">
<link href="assets/css/code.css" rel="stylesheet" type="text/css">
<link href="assets/css/theme.css" rel="stylesheet" type="text/css">

<link rel="alternate" type="application/rss+xml" title="RSS" href="rss.xml">
<link rel="canonical" href="https://example.com/index.html">
<!--[if lt IE 9]><script src="assets/js/html5.js"></script><![endif]--><link rel=
↪"prefetch" href="posts/welcome-to-nikola.html" type="text/html">
</head>
```

Luckily, since this is all under a very liberal license, we can just copy these CSS files into Nikola, adapting the paths a little so that they follow our conventions:

```
$ mkdir -p themes/lanyon/assets/css
$ cp ../lanyon/public/css/poole.css themes/lanyon/assets/css/
$ cp ../lanyon/public/css/lanyon.css themes/lanyon/assets/css/
```

Notice I am *not* copying `syntax.css`? That's because Nikola handles that styles for syntax highlighting in a particular way, using a setting called `CODE_COLOR_SCHEME` where you can configure what color scheme the syntax highlighter uses. You can use your own `assets/css/code.css` if you don't like the provided ones.

Nikola **requires** `assets/css/html4css1.css` and `assets/css/code.css` to function properly. We will also add themes for Jupyter (`assets/css/ipython.min.css` and `assets/css/nikola_ipython.css`) into the template; note that they are activated only if you configured your `POSTS`/`PAGES` with ipynb support. There's also `assets/css/nikola_rst.css`, which adds Bootstrap 3-style reST notes etc.

But how do I tell **our** lanyon theme to use those CSS files instead of whatever it's using now? By giving our theme its own base_helper.tmpl.

That file is a **template** used to generate parts of the pages. It's large and complicated but we don't need to change a lot of it. First, make a copy in your theme (note this command requires setting your `THEME` in `conf.py` to `lanyon`):

```
$ nikola theme -c base_helper.tmpl
```

The part we want to change is this:

```
<%def name="html_stylesheets()">
    %if use_bundles:
        %if use_cdn:
            <link href="/assets/css/all.css" rel="stylesheet" type="text/css">
        %else:
            <link href="/assets/css/all-nocdn.css" rel="stylesheet" type="text/css">
        %endif
    %else:
        <link href="/assets/css/html4css1.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/nikola_rst.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/code.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/theme.css" rel="stylesheet" type="text/css">
        %if has_custom_css:
            <link href="/assets/css/custom.css" rel="stylesheet" type="text/css">
        %endif
    %endif
    % if needs_ipython_css:
        <link href="/assets/css/ipython.min.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/nikola_ipython.css" rel="stylesheet" type="text/css">
    % endif
</%def>
```

And we will change it so it uses the lanyon styles instead of theme.css (again, ignore the bundles for now!):

```
<%def name="html_stylesheets()">
    %if use_bundles:
        <link href="/assets/css/all.css" rel="stylesheet" type="text/css">
    %else:
        <link href="/assets/css/html4css1.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/nikola_rst.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/poole.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/lanyon.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/code.css" rel="stylesheet" type="text/css">
        %if has_custom_css:
            <link href="/assets/css/custom.css" rel="stylesheet" type="text/css">
        %endif
    %endif
    % if needs_ipython_css:
        <link href="/assets/css/ipython.min.css" rel="stylesheet" type="text/css">
        <link href="/assets/css/nikola_ipython.css" rel="stylesheet" type="text/css">
    % endif
    <link rel="stylesheet" href="http://fonts.googleapis.com/css?family=PT+Serif:400,
↪400italic,700|PT+Sans:400">
</%def>
```

Fig. 3.2: You may say this looks like crap. Don't worry, we are just starting :-)

Page Layout

This is trickier but should be no problem for people with a basic understanding of HTML and a desire to make a theme!

Lanyon's content is split in two parts: a sidebar and the rest. The sidebar looks like this (shortened for comprehension):

```
<body>
<!-- Target for toggling the sidebar `.sidebar-checkbox` is for regular
     styles, `#sidebar-checkbox` for behavior. -->
<input type="checkbox" class="sidebar-checkbox" id="sidebar-checkbox">

<!-- Toggleable sidebar -->
<div class="sidebar" id="sidebar">
    <div class="sidebar-item">
        <p>A reserved <a href="http://jekyllrb.com" target="_blank">Jekyll</a> theme␣
↪that places the utmost gravity on content with a hidden drawer. Made by <a href=
↪"https://twitter.com/mdo" target="_blank">@mdo</a>.</p>
    </div>

    <nav class="sidebar-nav">
        <a class="sidebar-nav-item active" href="/">Home</a>
        <a class="sidebar-nav-item" href="/about/">About</a>
        [...]
    </nav>
</div>
```

So, a plain body, with an input element that controls the sidebar, a div which is the sidebar itself. Inside that, div.sidebar-item for items, and a nav with "navigational links". This is followed by the "masthead" and the content itself, which we will look at in a bit.

If we look for the equivalent code in Nikola's side, we see this:

```
<body>
<a href="#content" class="sr-only sr-only-focusable">Skip to main content</a>
<div id="container">
<header id="header" role="banner">
<h1 id="brand"><a href="https://example.com/" title="My Nikola Site" rel="home">
↪<span id="blog-title">My Nikola Site</span> </a></h1>
<nav id="menu" role="navigation"><ul>
<li><a href="../archive.html">Archive</a></li>
                <li><a href="../categories/index.html">Tags</a></li>
                <li><a href="../rss.xml">RSS feed</a></li>
```

So Nikola has the "masthead" above the nav element, and uses list elements in nav instead of bare links. Not all that different is it?

Let's make it lanyon-like! We will need 2 more templates: base.tmpl and base_header.tmpl. Get them and put them in your `themes/lanyon/templates` folder.

Let's look at `base.tmpl` first. It's short and nice, it looks like a webpage without all the interesting stuff:

```
## -*- coding: utf-8 -*-
<%namespace name="base" file="base_helper.tmpl" import="*"/>
<%namespace name="header" file="base_header.tmpl" import="*"/>
<%namespace name="footer" file="base_footer.tmpl" import="*"/>
${set_locale(lang)}
${base.html_headstart()}
<%block name="extra_head">
### Leave this block alone.
</%block>
${template_hooks['extra_head']()}
</head>
<body>
<a href="#content" class="sr-only sr-only-focusable">${messages("Skip to main content
↪")}</a>
```

```
    <div id="container">
        ${header.html_header()}
        <main id="content" role="main">
            <%block name="content"></%block>
        </main>
        ${footer.html_footer()}
    </div>
    ${body_end}
    ${template_hooks['body_end']()}
    ${base.late_load_js()}
</body>
</html>
```

That link which says "Skip to main content" is very important for accessibility, so we will leave it in place. But below, you can see how it creates the "container" div we see in the Nikola page, and the content is created by `html_header()` which is defined in `base_header.tmpl` The actual `nav` element is done by the `html_navigation_links` function out of the `NAVIGATION_LINKS` option.

So, first, lets change that base template to be more lanyon-like:

```
## -*- coding: utf-8 -*-
<%namespace name="base" file="base_helper.tmpl" import="*"/>
<%namespace name="header" file="base_header.tmpl" import="*"/>
<%namespace name="footer" file="base_footer.tmpl" import="*"/>
${set_locale(lang)}
${base.html_headstart()}
<%block name="extra_head">
### Leave this block alone.
</%block>
${template_hooks['extra_head']()}
</head>
<body>
    <a href="#content" class="sr-only sr-only-focusable">${messages("Skip to main
↪content")}</a>
    <!-- Target for toggling the sidebar `.sidebar-checkbox` is for regular
            styles, `#sidebar-checkbox` for behavior. -->
    <input type="checkbox" class="sidebar-checkbox" id="sidebar-checkbox">

    <!-- Toggleable sidebar -->
    <div class="sidebar" id="sidebar">
        <div class="sidebar-item">
            <p>A reserved <a href="http://getnikola.com" target="_blank">Nikola</a>
↪theme that places the utmost gravity on content with a hidden drawer. Made by <a
↪href="https://twitter.com/mdo" target="_blank">@mdo</a> for Jekyll,
            ported to Nikola by <a href="https://twitter.com/ralsina" target="_blank">
↪@ralsina</a>.</p>
        </div>
        ${header.html_navigation_links()}
    </div>

    <main id="content" role="main">
        <%block name="content"></%block>
    </main>
    ${footer.html_footer()}
    ${body_end}
    ${template_hooks['body_end']()}
    ${base.late_load_js()}
</body>
```

```
</html>
```

Fig. 3.3: And that's after I exposed the sidebar by clicking on an invisible widget!

One problem, which causes that yellow color in the sidebar is a CSS conflict. We are loading `html4css1.css` which specifies the background color of `div.sidebar` which is more specific than `lanyon.css`, which specifies for `.sidebar` alone.

There are many ways to fix this, I chose to change lanyon.css to also use div.sidebar:

```
div.sidebar,.sidebar {
    position: fixed;
    top: 0;
    bottom: 0;
    left: -14rem;
    width: 14rem;
    [...]
```

This is annoying but it will happen when you just grab CSS from different places. The "Inspect Element" feature of your web browser is your best friend for these situations.

Another problem is that the contents of the nav element are wrong. They are not bare links. We will fix that in `base_header.html`, like this:

```
<%def name="html_navigation_links()">
    <nav id="menu" role="navigation" class="sidebar-nav">
    %for url, text in navigation_links[lang]:
        <a class="sidebar-nav-item" href="${url}">${text}</a>
    %endfor
    ${template_hooks['menu']()}
    ${template_hooks['menu_alt']()}
    </nav>
</%def>
```

Note: this means this theme will not support submenus in navigation. If you want that, I'll happily take a patch.

Fig. 3.4: Starting to see a resemblance?

Now let's look at the content. In Lanyon, this is how the "main" content looks:

```
<!-- Wrap is the content to shift when toggling the sidebar. We wrap the
     content to avoid any CSS collisions with our real content. -->
<div class="wrap">
  <div class="masthead">
    <div class="container">
      <h3 class="masthead-title">
        <a href="/" title="Home">Lanyon</a>
        <small>A Jekyll theme</small>
      </h3>
    </div>
  </div>

  <div class="container content">
    <div class="post">
```

```
        <h1 class="post-title">Introducing Lanyon</h1>
        <span class="post-date">02 Jan 2014</span>
        <p>Lanyon is an unassuming <a href="http://jekyllrb.com">Jekyll</a> theme [...
↪]
    </div>
  </div>
</div>
<label for="sidebar-checkbox" class="sidebar-toggle"></label>
</body>
</html>
```

Everything inside the "container content" div is... the content. The rest is a masthead with the site title and at the bottom a label for the sidebar toggle. Easy to do in `base.tmpl` (only showing the relevant part):

```
    <!-- Wrap is the content to shift when toggling the sidebar. We wrap the
        content to avoid any CSS collisions with our real content. -->
    <div class="wrap">
    <div class="masthead">
        <div class="container">
        <h3 class="masthead-title">
            <a href="/" title="Home">Lanyon</a>
            <small>A Jekyll theme</small>
        </h3>
        </div>
    </div>

    <div class="container content" id="content">
        <%block name="content"></%block>
    </div>
    </div>
    <label for="sidebar-checkbox" class="sidebar-toggle"></label>
    ${footer.html_footer()}
    ${body_end}
    ${template_hooks['body_end']()}
    ${base.late_load_js()}
</body>
</html>
```

Fig. 3.5: Getting there!

The sidebar looks bad because of yet more CSS conflicts with `html4css1.css`. By adding some extra styling in `lanyon.css`, it will look better.

```
/* Style and "hide" the sidebar */
div.sidebar, .sidebar {
  position: fixed;
  top: 0;
  bottom: 0;
  left: -14rem;
  width: 14rem;
  visibility: hidden;
  overflow-y: auto;
  padding: 0;
  margin: 0;
  border: none;
  font-family: "PT Sans", Helvetica, Arial, sans-serif;
  font-size: .875rem; /* 15px */
```

```
  color: rgba(255,255,255,.6);
  background-color: #202020;
  -webkit-transition: all .3s ease-in-out;
          transition: all .3s ease-in-out;
}
```

Also, the accessibility link on top is visible when it should not. That's because we removed `theme.css` from the base theme, and with it, we lost a couple of classes. We can add them in `lanyon.css`, along with others used by other pieces of the site:

```
.sr-only {
  position: absolute;
  width: 1px;
  height: 1px;
  padding: 0;
  margin: -1px;
  overflow: hidden;
  clip: rect(0, 0, 0, 0);
  border: 0;
}

.sr-only-focusable:active,
.sr-only-focusable:focus {
  position: static;
  width: auto;
  height: auto;
  margin: 0;
  overflow: visible;
  clip: auto;
}

.breadcrumb {
  padding: 8px 15px;
  margin-bottom: 20px;
  list-style: none;
}

.breadcrumb > li {
  display: inline-block;
  margin-right: 0;
  margin-left: 0;
}

.breadcrumb > li:after {
  content: ' / ';
  color: #888;
}

.breadcrumb > li:last-of-type:after {
  content: '';
  margin-left: 0;
}

.thumbnails > li {
  display: inline-block;
  margin-right: 10px;
}
```

```
.thumbnails > li:last-of-type {
  margin-right: 0;
}
```

Fig. 3.6: Little by little, things look better.

One clear problem is that the title "Lanyon · A Jekyll theme" is set in the theme itself. We don't do that sort of thing in Nikola, we have settings for that. So, let's use them. There is a `html_site_title` function in `base_helper.tmpl` which is just the thing. So we change base.tmpl to use it:

```
<div class="wrap">
  <div class="masthead">
    <div class="container">
      ${header.html_site_title()}
    </div>
  </div>
```

That's a `<h1>` instead of a `<h3>` like Lanyon does, but hey, it's the right thing to do. If you want to go with an `<h3>`, just change `html_site_title` itself.

And now we more or less have the correct page layout and styles. Except for a rather large thing...

Typography

You can see in the previous screenshot that text still looks quite different in our port: Serif versus Sans-Serif content, and the titles have different colors!

Let's start with the titles. Here's how they look in Lanyon:

```
<h3 class="masthead-title">
  <a href="/" title="Home">Lanyon</a>
  <small>A Jekyll theme</small>
</h3>
```

Versus our port:

```
<h1 id="brand"><a href="https://example.com/" title="My Nikola Site" rel="home">
```

So, it looks like we will have to fix `html_site_title` after all:

```
<%def name="html_site_title()">
    <h3 id="brand" class="masthead-title">
    <a href="${abs_link(_link("root", None, lang))}" title="${blog_title}" rel="home">
↪${blog_title}</a>
    </h3>
</%def>
```

As for the actual content, that's not in any of the templates we have seen so far. The page you see is an "index.tmpl" page, which means it's a list of blog posts shown one below the other. Obviously it's not doing things in the way the Lanyon CSS expects it to. Here's the original, which you can find in Nikola's source code:

```
## -*- coding: utf-8 -*-
<%namespace name="helper" file="index_helper.tmpl"/>
<%namespace name="comments" file="comments_helper.tmpl"/>
```

```
<%inherit file="base.tmpl"/>

<%block name="extra_head">
    ${parent.extra_head()}
    % if posts and (permalink == '/' or permalink == '/' + index_file):
        <link rel="prefetch" href="${posts[0].permalink()}" type="text/html">
    % endif
</%block>

<%block name="content">
<%block name="content_header"></%block>
<div class="postindex">
% for post in posts:
    <article class="h-entry post-${post.meta('type')}">
    <header>
        <h1 class="p-name entry-title"><a href="${post.permalink()}" class="u-url">$
↪{post.title()|h}</a></h1>
        <div class="metadata">
            <p class="byline author vcard"><span class="byline-name fn">${post.
↪author()}</span></p>
            <p class="dateline"><a href="${post.permalink()}" rel="bookmark"><time␣
↪class="published dt-published" datetime="${post.date.isoformat()}" title="${post.
↪formatted_date(date_format)}">${post.formatted_date(date_format)}</time></a></p>
            % if not post.meta('nocomments') and site_has_comments:
                <p class="commentline">${comments.comment_link(post.permalink(), post.
↪_base_path)}
            % endif
        </div>
    </header>
    %if index_teasers:
    <div class="p-summary entry-summary">
    ${post.text(teaser_only=True)}
    %else:
    <div class="e-content entry-content">
    ${post.text(teaser_only=False)}
    %endif
    </div>
    </article>
% endfor
</div>
${helper.html_pager()}
${comments.comment_link_script()}
${helper.mathjax_script(posts)}
</%block>
```

And this is how it looks after I played with it for a while, making it generate code that looks closer to the Lanyon original:

```
<%block name="content">
<%block name="content_header"></%block>
<div class="posts">
% for post in posts:
    <article class="post h-entry post-${post.meta('type')}">
    <header>
        <h1 class="post-title p-name"><a href="${post.permalink()}" class="u-url">$
↪{post.title()|h}</a></h1>
        <div class="metadata">
            <p class="byline author vcard"><span class="byline-name fn">${post.
↪author()}</span></p>
```

```
            <p class="dateline"><a href="${post.permalink()}" rel="bookmark"><time␣
↪class="post-date published dt-published" datetime="${post.date.isoformat()}" title="
↪${post.formatted_date(date_format)}">${post.formatted_date(date_format)}</time></a>
↪</p>
            % if not post.meta('nocomments') and site_has_comments:
                <p class="commentline">${comments.comment_link(post.permalink(), post.
↪_base_path)}
            % endif
        </div>
    </header>
    %if index_teasers:
    <div class="p-summary entry-summary">
    ${post.text(teaser_only=True)}
    %else:
    <div class="e-content entry-content">
    ${post.text(teaser_only=False)}
    %endif
    </div>
    </article>
% endfor
</div>
${helper.html_pager()}
${comments.comment_link_script()}
${helper.mathjax_script(posts)}
</%block>
```

With these changes, it looks… similar?

Fig. 3.7: It does!

Similar changes (basically adding class names to elements) needed to be done in `post_header.tmpl`:

```
<%def name="html_post_header()">
    <header>
        ${html_title()}
        <div class="metadata">
            <p class="byline author vcard"><span class="byline-name fn">${post.
↪author()}</span></p>
            <p class="dateline"><a href="${post.permalink()}" rel="bookmark"><time␣
↪class="post-date published dt-published" datetime="${post.date.isoformat()}"␣
↪itemprop="datePublished" title="${post.formatted_date(date_format)}">${post.
↪formatted_date(date_format)}</time></a></p>
            % if not post.meta('nocomments') and site_has_comments:
                <p class="commentline">${comments.comment_link(post.permalink(), post.
↪_base_path)}
            % endif
            %if post.description():
                <meta name="description" itemprop="description" content="${post.
↪description()}">
            %endif
        </div>
        ${html_translations(post)}
    </header>
</%def>
```

Customization

The original Lanyon theme supports some personalization options. It suggests you do them by tweaking the templates, and you *can* also do that in the Nikola port. But we prefer to use options for that, so that you can get a later, better version of the theme and it will still "just work".

Let's see the color schemes first. They apply easily, just tweak your `body` element like this:

```
<body class="theme-base-08">
...
</body>
```

We can tweak `base.tmpl` to do just that:

```
% if lanyon_subtheme:
<body class="${lanyon_subtheme}">
%else:
<body>
%endif
```

And then we can put the options in conf.py's `GLOBAL_CONTEXT`:

```
GLOBAL_CONTEXT = {
    "lanyon_subtheme": "theme-base-08"
}
```

Fig. 3.8: Look at it, all themed up.

Doing the same for layout-reverse, sidebar-overlay and the rest is left as an exercise for the reader.

Bundles

If you have `webassets` installed and the `USE_BUNDLES` option set to True, Nikola can put several CSS or JS files together in a larger file, which makes sites load faster. To do that, your theme needs a `bundles` file where the syntax is:

```
outputfile1.js=thing1.js,thing2.js,...
outputfile2.css=thing1.css,thing2.css,...
```

For the Lanyon theme, it should be like this:

```
assets/css/all.css=html4css1.css,nikola_rst.css,code.css,poole.css,lanyon.css,custom.
↪css
```

Note: Some themes also support the `USE_CDN` option meaning that in some cases it will load one bundle with all CSS and in other will load some CSS files from a CDN and others from a bundle. This is complicated and probably not worth the effort.

The End

And that's it, that's a whole theme. Eventually, once people start using it, they will notice small broken details, which will need handling one at a time.

This theme should be available in http://themes.getnikola.com/v7/lanyon/ and you can see it in action at https://themes.getnikola.com/v7/lanyon/demo/ .

CHAPTER 4

Theming Nikola

Version 7.8.8

Author Roberto Alsina <ralsina@netmanagers.com.ar>

Contents

This document is a reference about themes. If you want a tutorial, please read *Creating a Theme*. If you're looking for a ready-made theme for your site, check out the Themes Index.

The Structure

Themes are located in the `themes` folder where Nikola is installed, and in the `themes` folder of your site, one folder per theme. The folder name is the theme name.

A Nikola theme consists of the following folders (they are *all* optional):

assets This is where you would put your CSS, JavaScript and image files. It will be copied into `output/assets` when you build the site, and the templates will contain references to them. The default subdirectories are `css`, `js`, `xml` and `fonts` (Bootstrap).

The included themes use Bootstrap, Colorbox, Flowr.js and Moment.js, so they are in assets, along with CSS files for syntax highlighting, reStructuredText and Jupyter, as well as a minified copy of jQuery.

If you want to base your theme on other frameworks (or on no framework at all) just remember to put there everything you need for deployment. (Not all of the listed assets are used by `base`)

templates This contains the templates used to generate the pages. While Nikola will use a certain set of template names by default, you can add others for specific parts of your site.

messages Nikola tries to be multilingual. This is where you put the strings for your theme so that it can be translated into other languages.

less, sass Files to be compiled into CSS using LESS and Sass (both require plugins)

This mandatory file:

<theme>.theme An INI file containing theme meta data. The file format is described in detail below, in the *Theme meta files* section.

And these optional files:

parent, engine One-line text files that contain the names of parent and engine themes, respectively. Those are needed for older versions (Nikola v7.8.5 and older).

bundles A text file containing a list of files to be turned into bundles using WebAssets. For example:

```
assets/css/all.css=bootstrap.css,html4css1.css,nikola_rst.css,code.css,colorbox.
↪css,custom.css
```

This creates a file called "assets/css/all.css" in your output that is the combination of all the other file paths, relative to the output file. This makes the page much more efficient because it avoids multiple connections to the server, at the cost of some extra difficult debugging.

WebAssets supports bundling CSS and JS files.

Templates should use either the bundle or the individual files based on the `use_bundles` variable, which in turn is set by the `USE_BUNDLES` option.

Theme meta files

As of Nikola v7.8.6, Nikola uses meta files for themes. Those are INI files, with the same name as your theme, and a `.theme` extension, eg. `bootstrap3.theme`. Here is an example, from the bootstrap3 theme:

```
[Theme]
engine = mako
parent = base
author = The Nikola Contributors
author_url = https://getnikola.com/
based_on = Bootstrap 3 <http://getbootstrap.com/>
license = MIT
tags = bootstrap

[Family]
family = bootstrap3
jinja_version = bootstrap3-jinja
variants = bootstrap3-gradients, bootstrap3-gradients-jinja
```

```
[Nikola]
bootswatch = True
```

The following keys are currently supported:

- `Theme` — contains information about the theme.
  - `engine` — engine used by the theme. Should be `mako` or `jinja`.
  - `parent` — the parent theme. Any resources missing in this theme, will be looked up in the parent theme (and then in the grandparent, etc).

    The parent is so you don't have to create a full theme each time: just create an empty theme, set the parent, and add the bits you want modified. You **must** define a parent, otherwise many features won't work due to missing templates, messages, and assets.

    The following settings are recommended:

    * If your theme uses Bootstrap 3, inherit the `bootstrap3` theme.
    * If your theme uses Jinja as a template engine, inherit `base-jinja` or `bootstrap3-jinja`
    * In any other case, inherit `base`.
  - `author`, `author_url` — used to identify theme author.
  - `based_on` — optional list of inspirations, frameworks, etc. used in the theme. Should be comma-separated, the format `Name <URL>` is recommended.
  - `license` — theme license. Pick MIT if you have no preference.
  - `tags` — optional list of tags (comma-separated) to describe the theme.
- `Family` — contains information about other related themes. All values optional.
  - `family` — the name of the main theme in a family, which is also used as the family name.
  - `mako_version`, `jinja_version` — name of the mako/jinja version of the theme.
  - `variants` — comma-separated list of stylistic variants (other than the mako/jinja version listed above)
- `Nikola` — Nikola-specific information, currently optional.
  - `bootswatch` — whether or not theme supports Bootswatch styling (optional, defaults to False)
  - `ignored_assets` — comma-separated list of assets to ignore (relative to the `assets/` directory, eg. `css/theme.css`)
  - `ignore_colorbox_i18n` — prevent copying Colorbox locales. Accepted values: `all` (all ignored), `unused` (used locales copied), `none` (all copied)

Templates

In templates there is a number of files whose name ends in `.tmpl`. Those are the theme's page templates. They are done using the Mako or Jinja2 template languages. If you want to do a theme, you should learn one first. What engine is used by the theme is declared in the `engine` file.

Tip: If you are using Mako templates, and want some extra speed when building the site you can install Beaker and make templates be cached

Both template engines have a nifty concept of template inheritance. That means that, a template can inherit from another and only change small bits of the output. For example, `base.tmpl` defines the whole layout for a page but has only a placeholder for content so `post.tmpl` only define the content, and the layout is inherited from `base.tmpl`.

These are the templates that come with the included themes:

`base.tmpl` This template defines the basic page layout for the site. It's mostly plain HTML but defines a few blocks that can be re-defined by inheriting templates.

It has some separate pieces defined in `base_helper.tmpl`, `base_header.tmpl` and `base_footer.tmpl` so they can be easily overridden.

`index.tmpl` Template used to render the multipost indexes. The posts are in a `posts` variable. Some functionality is in the `index_helper.tmpl` helper template.

`archive_navigation_helper.tmpl` Code that implements archive navigation (previous/up/next). Included by archive templates.

`archiveindex.tmpl` Used to display archives, if `ARCHIVES_ARE_INDEXES` is True. By default, it just inherits `index.tmpl`, with added archive navigation and feeds.

`author.tmpl` Used to display author pages.

`authorindex.tmpl` Used to display author indexes, if `AUTHOR_PAGES_ARE_INDEXES` is True. By default, it just inherits `index.tmpl`, with added feeds.

`comments_helper.tmpl` (internal) This template handles comments. You should probably never touch it :-) It uses a bunch of helper templates, one for each supported comment system (all of which start with `comments_helper`)

`crumbs.tmpl`, `slides.tmpl`, `pagination_helper.tmpl` These templates help render specific UI items, and can be tweaked as needed.

`gallery.tmpl` Template used for image galleries. Interesting data includes:

- `post`: A post object, containing descriptive `post.text()` for the gallery.
- `crumbs`: A list of `link, crumb` to implement breadcrumbs.
- `folders`: A list of folders to implement hierarchical gallery navigation.
- `enable_comments`: To enable/disable comments in galleries.
- `thumbnail_size`: The `THUMBNAIL_SIZE` option.
- `photo_array`: a list of dictionaries, each containing:
  - `url`: URL for the full-sized image.
  - `url_thumb`: URL for the thumbnail.
  - `title`: The title of the image.
  - `size`: A dict containing `w` and `h`, the real size of the thumbnail.
- `photo_array_json`: a JSON dump of photo_array, used in the bootstrap theme by flowr.js

`list.tmpl` Template used to display generic lists of links, which it gets in `items`, a list of *(text, link, count)* elements.

`list_post.tmpl` Template used to display generic lists of posts, which it gets in `posts`.

`listing.tmpl` Used to display code listings.

`math_helper.tmpl` (internal) Used to add MathJax/KaTeX code to pages.

post.tmpl Template used by default for blog posts, gets the data in a `post` object which is an instance of the Post class. Some functionality is in the `post_helper.tmpl` and `post_header.tmpl` templates.

post_list_directive.tmpl Template used by the `post_list` reStructuredText directive.

sectionindex.tmpl Used to display section indexes, if `POST_SECTIONS_ARE_INDEXES` is True. By default, it just inherits `index.tmpl`, with added feeds.

story.tmpl Used for pages that are not part of a blog, usually a cleaner, less intrusive layout than `post.tmpl`, but same parameters.

tag.tmpl Used to show the contents of a single tag or category.

tagindex.tmpl Used to show the contents of a single tag or category, if `TAG_PAGES_ARE_INDEXES` is True. By default, it just inherits `index.tmpl`, with added feeds and some extra features.

tags.tmpl Used to display the list of tags and categories.

You can add other templates for specific pages, which the user can then use in his `POSTS` or `PAGES` option in `conf.py`. Also, keep in mind that your theme is *yours*, there is no reason why you would need to maintain the inheritance as it is, or not require whatever data you want (eg. you may depend on specific custom `GLOBAL_CONTEXT` variables, or post meta attributes)

Also, you can specify a custom template to be used by a post or page via the `template` metadata, and custom templates can be added in the `templates/` folder of your site.

Variables available in templates

The full, complete list of variables available in templates is maintained in a separate document: Template variables

Customizing themes to user color preference and section colors

The user's preference for theme color is exposed in templates as `theme_color` set in the `THEME_COLOR` option.

Each section has an assigned color that is either set by the user or auto selected by adjusting the hue of the user's `THEME_COLOR`. The color is exposed in templates through `post.section_color(lang)`. The function that generates the colors from strings and any given color (by section name and theme color for sections) is exposed through the `colorize_str_from_base_color(string, hex_color)` function

Hex color values, like that returned by the theme or section color can be altered in the HSL colorspace through the function `color_hsl_adjust_hex(hex_string, adjust_h, adjust_s, adjust_l)`. Adjustments are given in values between 1.0 and -1.0. For example, the theme color can be made lighter using this code:

```
<!-- Mako -->
<span style="color: ${color_hsl_adjust_hex(theme_color, adjust_l=0.05)}">
```

```
<!-- Jinja2 -->
<span style="color: {{ color_hsl_adjust_hex(theme_color, adjust_l=0.05) }}">
```

Identifying and customizing different kinds of pages with a shared template

Nikola provides a `pagekind` in each template contexts that can be used to modify shared templates based on the context it's being used. For example, the `base_helper.tmpl` is used in all pages, `index.tmpl` is used in many contexts and you may want to add or remove something from only one of these contexts.

Example of conditionally loading different resources on all index pages (archives, author pages, and tag pages), and others again to the front page and in every post pages:

```
<!-- Mako -->
<head>
    ...
    % if 'index' in pagekind:
        <link href="/assets/css/multicolumn.css" rel="stylesheet">
    % endif
    % if 'front_page' in pagekind:
        <link href="/assets/css/fancy_homepage.css" rel="stylesheet">
        <script src="/assets/js/post_carousel.js"></script>
    % endif
    % if 'post_page' in pagekind:
        <link href="/assets/css/article.css" rel="stylesheet">
        <script src="/assets/js/comment_system.js"></script>
    % endif
</head>
```

```
<!-- Jinja2 -->
<head>
    ...
    {% if 'index' in pagekind %}
        <link href="/assets/css/multicolumn.css" rel="stylesheet">
    {% endif %}
    {% if 'front_page' in pagekind %}
        <link href="/assets/css/fancy_homepage.css" rel="stylesheet">
        <script src="/assets/js/post_carousel.js"></script>
    {% endif %}
    {% if 'post_page' in pagekind %}
        <link href="/assets/css/article.css" rel="stylesheet">
        <script src="/assets/js/comment_system.js"></script>
    {% endif %}
</head>
```

Promoting visits to the front page when visiting other filtered `index.tmpl` page variants such as author pages and tag pages. This could have been included in `index.tmpl` or maybe in `base.tmpl` depending on what you want to achieve.

```
<!-- Mako -->
% if 'index' in pagekind:
    % if 'author_page' in pagekind:
        <p>These posts were written by ${author}. See posts by all
           authors on the <a href="/">front page</a>.</p>
    % elif 'tag_page' in pagekind:
        <p>This is a filtered selection of posts tagged “${tag}”, visit
           the <a href="/">front page</a> to see all posts.</p>
    % endif
% endif
```

```
<!-- Jinja2 -->
{% if 'index' in pagekind %}
    {% if 'author_page' in pagekind %}
        <p>These posts were written by {{ author }}. See posts by all
           authors on the <a href="/">front page</a>.</p>
    {% elif 'tag_page' in pagekind %}
        <p>This is a filtered selection of posts tagged “{{ tag }}”, visit
           the <a href="/">front page</a> to see all posts.</p>
    {% endif %}
{% endif %}
```

List of page kinds provided by default plugins:

- front_page
- index
- index, archive_page
- index, author_page
- index, main_index
- index, section_page
- index, tag_page
- list
- list, archive_page
- list, author_page
- list, section_page
- list, tag_page
- list, tags_page
- post_page
- page_page
- story_page
- listing
- generic_page
- gallery_front
- gallery_page

Messages and Translations

The included themes are translated into a variety of languages. You can add your own translation at https://www.transifex.com/projects/p/nikola/

If you want to create a theme that has new strings, and you want those strings to be translatable, then your theme will need a custom `messages` folder.

LESS and Sass

Note: The LESS and Sass compilers were moved to the Plugins Index in Nikola v7.0.0.

If you want to use those CSS extensions, you can — just store your files in the `less` or `sass` directory of your theme.

In order to have them work, you need to create a list of `.less` or `.scss/.sass` files to compile — the list should be in a file named `targets` in the respective directory (`less`/`sass`).

The files listed in the `targets` file will be passed to the respective compiler, which you have to install manually (`lessc` which comes from the Node.js package named `less` or `sass` from a Ruby package aptly named `sass`). Whatever the compiler outputs will be saved as a CSS file in your rendered site, with the `.css` extension.

Note: Conflicts may occur if you have two files with the same base name but a different extension. Pay attention to how you name your files or your site won't build! (Nikola will tell you what's wrong when this happens)

Variables available in templates are listed below.

- This list is maintained by humans, so it may not always be perfect.
- Variables whose types are marked with `?` may not always be available or may be None in some cases.
- Templates usually do not have access to the original TranslatableSetting variables, only to the current locale version (except `NAVIGATION_LINKS`).
- For function and setting documentation, please consult code documentation and default configuration respectively.
- Templates often create their own functions (macros), and import macros from other templates. Those macros are not listed here.

Variables and functions come from three places:

- the global context
- the local context of a page
- the templates themselves and the templates they import

Contents

CHAPTER 5

Global variables

Some variables on the global variables list may be None (the ? symbol is not used).

Name	Type	Description
_link	function	Nikola.link function
abs_link	function	Nikola.abs_link function
author_pages_generated	bool	False
blog_author	TranslatableSetting<str>	BLOG_AUTHOR setting
blog_description	TranslatableSetting<str>	BLOG_DESCRIPTION setting
blog_title	TranslatableSetting<str>	BLOG_TITLE setting
blog_url	str	SITE_URL setting
body_end	TranslatableSetting<str>	BODY_END setting
colorbox_locales	defaultdict<str, str>	dictionary of available Colorbox locales
colorize_str_from_base_color	function	utils.colorize_str_from_base_color f
color_hsl_adjust_hex	function	utils.color_hsl_adjust_hex function
comment_system_id	str	COMMENT_SYSTEM_ID setting
comment_system	str	COMMENT_SYSTEM setting
content_footer	TranslatableSetting<str>	CONTENT_FOOTER setting
data	dict	data files (from the data/ directory)
date_fanciness	int	DATE_FANCINESS setting
date_format	TranslatableSetting<str>	DATE_FORMAT setting
exists	function	Nikola.file_exists function
extra_head_data	TranslatableSetting<str>	EXTRA_HEAD_DATA setting
favicons	tuple	FAVICONS setting
front_index_header	TranslatableSetting<str>	FRONT_INDEX_HEADER setting
generate_atom	bool	GENERATE_ATOM setting
generate_rss	bool	GENERATE_RSS setting
global_data	dict	alias for data
has_custom_css	bool	True if custom.css exists
hidden_authors	list<str>	HIDDEN_AUTHORS setting
hidden_categories	list<str>	HIDDEN_CATEGORIES setting

Table 5.1 – continued from previous page

Name	Type	Description
hidden_tags	list<str>	HIDDEN_TAGS setting
hide_sourcelink	bool	SHOW_SOURCELINK setting, negated
index_display_post_count	int	INDEX_DISPLAY_POST_COUNT setting
index_file	str	INDEX_FILE setting
js_date_format	TranslatableSetting<str>	JS_DATE_FORMAT setting
katex_auto_render	str	KATEX_AUTO_RENDER setting
license	TranslatableSetting<str>	LICENSE setting
logo_url	str	LOGO_URL setting
mathjax_config	str	MATHJAX_CONFIG setting
messages	dict<dict<str, str>>	translated messages ({language: {english:
meta_generator_tag	bool	META_GENERATOR_TAG setting
momentjs_locales	defaultdict<str, str>	dictionary of available Moment.js locales
navigation_links	TranslatableSetting	NAVIGATION_LINKS setting
needs_ipython_css	bool	whether or not Jupyter CSS is needed by this site
posts_sections	bool	POSTS_SECTIONS setting
posts_section_are_indexes	bool	POSTS_SECTIONS_ARE_INDEXES setting
posts_sections_are_indexes	bool	POSTS_SECTIONS_ARE_INDEXES setting
posts_section_colors	TranslatableSetting	POSTS_SECTION_COLORS setting
posts_section_descriptions	Tss	POSTS_SECTION_DESCRIPTIONS setting
posts_section_from_meta	bool	POSTS_SECTION_FROM_META setting
posts_section_name	TranslatableSetting<str>	POSTS_SECTION_NAME setting
posts_section_title	TranslatableSetting<str>	POSTS_SECTION_TITLE setting
rel_link	function	Nikola.rel_link function
rss_link	str	RSS_LINK setting
rss_path	TranslatableSetting<str>	RSS_PATH setting
search_form	TranslatableSetting<str>	SEARCH_FORM setting
set_locale	function	LocaleBorg.set_locale function (or None if
show_blog_title	bool	SHOW_BLOG_TITLE setting
show_sourcelink	bool	SHOW_SOURCELINK setting
site_has_comments	bool	whether or not a comment system is configured
SLUG_AUTHOR_PATH	bool	SLUG_AUTHOR_PATH setting
SLUG_TAG_PATH	bool	SLUG_TAG_PATH setting
social_buttons_code	TranslatableSetting<str>	SOCIAL_BUTTONS_CODE setting
sort_posts	function	utils.sort_posts function
template_hooks	dict<str, TemplateHookRegistry>	Template hooks registered by plugins
theme_color	str	THEME_COLOR setting
timezone	tzinfo	Timezone object (represents the configured timezone
translations	dict<str, str>	TRANSLATIONS setting
twitter_card	dict	TWITTER_CARD setting, defaults to an empty dicti
url_replacer	function	Nikola.url_replacer function
url_type	str	URL_TYPE setting
use_base_tag	bool	USE_BASE_TAG setting
use_bundles	bool	USE_BUNDLES setting
use_cdn	bool	USE_CDN setting
use_katex	bool	USE_KATEX setting
use_open_graph	bool	USE_OPEN_GRAPH setting, defaults to True
subtheme	str?	THEME_REVEAL_CONFIG_SUBTHEME setting (o
transition	str?	THEME_REVEAL_CONFIG_TRANSITION setting

CHAPTER 6

Per-page local variables

Those variables are available on all pages, but their contents are dependent on page contents.

Name	Type	Description
`description`	str	Description of the page
`is_rtl`	bool	Whether or not the language is left-to-right
`lang`	str	Current language
`pagekind`	list<str>	List of strings that identify the type of this page (docs)
`title`	str	Title of the page (taken from post, config, etc.)
`formatmsg`	function	Wrapper over `%` string formatting
`striphtml`	function	Strips HTML tags (Mako only)

CHAPTER 7

Variables available in post pages (`post.tmpl`, `story.tmpl` etc.)

Name	Type	Description
`post`	Post	The post object
`permalink`	str	Permanent link to the post
`enable_comments`	bool	True for posts, `COMMENTS_IN_PAGES` setting for pages

CHAPTER 8

Variables available in post lists

Name	Type	Description
`posts`	list<Post>	List of post objects that appear in this list
`prevlink`	str	Link to previous page
`nextlink`	str	Link to next page

CHAPTER 9

Variables available in indexes

Name	Type	Description
`posts`	list<Post>	List of post objects that appear in this list
`index_teasers`	bool	`INDEX_TEASERS` setting
`show_index_page_navigation`	bool	`SHOW_INDEX_PAGE_NAVIGATION` setting
`current_page`	int	Number of current page
`page_links`	list<str>	Links to different pages
`prevlink`	str	Link to previous page
`nextlink`	str	Link to next page
`prevfeedlink`	str	Link to previous page as an Atom feed
`nextfeedlink`	str	Link to next page as an Atom feed
`prev_next_links_reversed`	bool	Whether or not previous and next links should be reversed (`INDEXES_STATIC`)

CHAPTER 10

Variables available in taxonomies

Variable names enclosed in <> are dependent on the taxonomy.

Taxonomy	Variable	Value
archive	overview_page_variable_name	archive
author	overview_page_variable_name	authors
category	overview_page_variable_name	categories
category	overview_page_items_variable_name	cat_items
category	overview_page_hierarchy_variable_name	cat_hierarchy
index	overview_page_variable_name	unavailable (None)
page_index_folder	overview_page_variable_name	page_folder
section_index	overview_page_variable_name	sections
tag	overview_page_variable_name	tags
tag	overview_page_items_variable_name	items

Templates and settings used by taxonomies

Taxonomy	Has hierarchy	List (one classification) template	Index (one classification) template	Overview (list of classifications) template	Subcategories list template	List is an index	Show as list of subcategories
(default settings)	no	tagindex.tmpl	tagindex.tmpl	list.tmpl	taxonomy_list.tmpl (does not exist)	no	no
`archive`	yes (0-3 levels)	list_post.tmpl	archiveindex.tmpl	list.tmpl	list.tmpl	`ARCHIVES_ARE_INDEXES`	`CREATE_FULL_ARCHIVES`
`author`	no	author.tmpl	authorindex.tmpl	authors.tmpl	n/a	`AUTHOR_PAGES_ARE_INDEXES`	no
`category`	yes	tag.tmpl	tagindex.tmpl	tags.tmpl (with tags)	n/a	`CATEGORY_PAGES_ARE_INDEXES`	no
`index`	no	n/a	index.tmpl	n/a	n/a	yes	no
`page_index`	yes	list.tmpl	n/a	n/a	n/a	no	no
`section_index`	no	list.tmpl	sectionindex.tmpl	n/a	n/a	`POSTS_SECTIONS_ARE_INDEXES`	no
`tag`	no	tag.tmpl	tagindex.tmpl	tags.tmpl (with categories)	n/a	`TAG_PAGES_ARE_INDEXES`	no

Classification overviews

Hierarchy-related variables are available if and only if `has_hierarchy` is True.

Name	Type	Description
`<overview_page_variable_name>`	str	List of classifications
`<overview_page_items_variable_name>`	list	List of items *(name, link)*
`<overview_page_items_variable_name + "_with_postcount">`	list	List of items *(name, link, number of posts)*
`<overview_page_hierarchy_variable_name>`	list?	List of hierarchies *(name, full name, path, link, indent levels, indent to change before, indent to change after)*
`<overview_page_hierarchy_variable_name + "_with_postcount">`	list?	List of hierarchies, with added counts *(name, full name, path, link, indent levels, indent to change before, indent to change after, number of children, number of posts)*
`has_hierarchy`	bool	Value of `has_hierarchy` for the taxonomy
`permalink`	str	Permanent link to page

Classification pages (lists)

Name	Type	Description
`kind`	str	The classification name
`items`	list?	List of items for `list.tmpl` *(title, permalink, None)*
`posts`	list<Post>?	List of items for other templates
`kind`	str	The classification name
`permalink`	str	Permanent link to page
`other_languages`	list<tuple>	List of triples `(other_lang, other_classification, title)`

Index-style classification pages have `kind` in addtion to the usual index variables.

Subclassification page

Name	Type	Description
`items`	list?	List of items
`permalink`	str	Permanent link to page
`other_languages`	list<tuple>	List of triples `(other_lang, other_classification, title)`

Hierarchical lists

The indenting information can be used to render the items as a tree. The values have the following meanings:

- `indent levels` is a list of pairs `(current_i, count_i)` giving the current position (`0`, ..., `count_i-1`) and maximum (`count_i`) in the hierarchy level `i`;
- `indent to change before` is the difference of hierarchy levels between the previous and the current item; positive values indicate that the current item is indented further in and can be used to open HTML tags before the item;
- `indent to change after` is the difference of hierarchy levels between the current and the next item; negative values indicate that the current item is indented further in and can be used to close HTML tags after the item.

Example:

```
+--- levels:[(0,3)], before:1, after:0
+-+- levels:[(1,3)], before:0, after:1
| +--- levels:[(1,3), (0,2)], before:1, after:0
| +-+- levels:[(1,3), (1,2)], before:0, after:1
|   +--- levels:[(1,3), (1,2), (0, 1)], before:1, after:-2
+-+- levels:[(2,3)], before:-2, after:1
  +- levels:[(2,3), (0,1)], before:1, after:-2
```

See `tags.tmpl` in the base themes for examples on how to render a tree as nested unordered lists in HTML.

CHAPTER 11

Variables available in archives

The archive navigation variables are available only if `create_archive_navigation` is True.

Name	Type	Description
`kind`	str	Always `"archive"`
`archive_name`	str?	Name of the archive (only if using indexes)
`create_archive_navigation`	bool	`CREATE_ARCHIVE_NAVIGATION` setting
`has_archive_navigation`	bool	Whether or not archive navigation is available
`up_archive`	str?	Link to the archive one level up
`up_archive_name`	str?	Name of the archive one level up
`previous_archive`	str?	Link to the previous archive
`previous_archive_name`	str?	Name of the previous archive
`next_archive`	str?	Link to the next archive
`next_archive_name`	str?	Name of the next archive
`archive_nodelevel`	int?	Level of the archive
`other_languages`	list	List of tuples `(lang, path, name)` of same archive in other languages

CHAPTER 12

Variables available in author pages

Name	Type	Description
`kind`	str	Always `"author"`
`author`	str	Author name
`rss_link`	str	Link to RSS (HTML fragment)
`other_languages`	list<tuple>	List of tuples `(lang, author, name)` of same author in other languages

CHAPTER 13

Variables available in category pages

Name	Type	Description
`kind`	str	Always "`category`"
`category`	str	Category name
`category_path`	list<str>	Category hierarchy
`rss_link`	str?	Link to RSS (HTML fragment, only if using indexes)
`subcategories`	list	List of subcategories (contains *name, link* tuples)
`tag`	str	Friendly category name
`other_languages`	list<tuple>	List of tuples `(lang, category, name)` of same category in other languages

CHAPTER 14

Variables available in galleries

Name	Type	Description
`crumbs`	list	Breadcrumbs for this page
`enable_comments`	bool	Whether or not comments are enabled in galleries
`folders`	list	List of folders (contains *path, title* tuples)
`permalink`	str	Permanent link to this page
`photo_array`	list	Photo array (contains dicts with image data: *url, url_thumb, title, size{w, h}*)
`photo_array_json`	str	Photo array in JSON format
`post`	Post?	The Post object for this gallery
`thumbnail_size`	int	`THUMBNAIL_SIZE` setting

CHAPTER 15

Variables available in listings

Name	Type	Description
`code`	str	Highlighted source code (HTML fragment)
`crumbs`	list	Breadcrumbs for this page
`folders`	list<str>	List of subfolders
`files`	list<str>	List of files in the folder
`source_link`	str	Link to the source file

CHAPTER 16

Variables available in sections

Name	Type	Description
`section`	str	Section name (internal)
`kind`	str	Always `"section"`
`other_languages`	list<tuple>	List of tuples `(lang, section, name)` of same section in other languages

CHAPTER 17

Variables available in tag pages

Name	Type	Description
`kind`	str	Always `"tag"`
`tag`	str	Tag name
`other_languages`	list<tuple>	List of tuples `(lang, tag, name)` of same tag in other languages

CHAPTER 18

Variables available in the “Tags and categories” page (`tags.tmpl`)

Name	Type	Description
`items`	list	Tags *(name, link)*
`cat_items`	list	Categories *(name, full name, path, link, indent levels, indent to change before, indent to change after)*

For more details about hierarchies, see *Hierarchical lists*

CHAPTER 19

Variables available in shortcodes

The global context is available in templated shortcodes.

Name	Type	Description
`lang`	str	Current language
`_args`	list<str>	Arguments given to the shortcode
`data`	str	Shortcode contents
`post`	Post	Post object (if available)
`filename`	str?	file name, if `shortcode_function.nikola_shortcode_pass_filename = True`

CHAPTER 20

Variables available in post lists

The global context is NOT available in post lists.

Name	Type	Description
`posts`	list<Post>	Posts that are on the list
`lang`	str	Current language
`date_format`	str	The date format for current language
`post_list_id`	str	GUID of post list
`messages`	dict	The messages dictionary
`_link`	function	`Nikola.link` function

CHAPTER 21

Extending Nikola

Version 7.8.8

Author Roberto Alsina <ralsina@netmanagers.com.ar>

Contents

Nikola is extensible. Almost all its functionality is based on plugins, and you can add your own or replace the provided ones.

Plugins consist of a metadata file (with `.plugin` extension) and a Python module (a `.py` file) or package (a folder containing a `__init__.py` file.

To use a plugin in your site, you just have to put it in a `plugins` folder in your site.

Plugins come in various flavours, aimed at extending different aspects of Nikola.

Command Plugins

When you run `nikola --help` you will see something like this:

```
$ nikola help
Nikola is a tool to create static websites and blogs. For full documentation and more
information, please visit https://getnikola.com/


Available commands:
nikola auto                  automatically detect site changes, rebuild
                             and optionally refresh a browser
nikola bootswatch_theme      given a swatch name from bootswatch.com and a
                             parent theme, creates a custom theme
nikola build                 run tasks
nikola check                 check links and files in the generated site
nikola clean                 clean action / remove targets
nikola console               start an interactive python console with access to
                             your site and configuration
nikola deploy                deploy the site
nikola dumpdb                dump dependency DB
nikola forget                clear successful run status from internal DB
nikola help                  show help
nikola ignore                ignore task (skip) on subsequent runs
nikola import_blogger        import a blogger dump
nikola import_feed           import a RSS/Atom dump
nikola import_wordpress      import a WordPress dump
nikola init                  create a Nikola site in the specified folder
nikola list                  list tasks from dodo file
nikola mincss                apply mincss to the generated site
nikola new_post              create a new blog post or site page
nikola run                   run tasks
nikola serve                 start the test webserver
nikola strace                use strace to list file_deps and targets
nikola theme                 manage themes
nikola version               print the Nikola version number

nikola help                  show help / reference
nikola help <command>        show command usage
nikola help <task-name>      show task usage
```

That will give you a list of all available commands in your version of Nikola. Each and every one of those is a plugin. Let's look at a typical example:

First, the `serve.plugin` file:

```
[Core]
Name = serve
```

```
Module = serve

[Documentation]
Author = Roberto Alsina
Version = 0.1
Website = https://getnikola.com
Description = Start test server.
```

Note: If you want to publish your plugin on the Plugin Index, read the docs for the Index (and the .plugin file examples and explanations).

For your own plugin, just change the values in a sensible way. The `Module` will be used to find the matching Python module, in this case `serve.py`, from which this is the interesting bit:

```
from nikola.plugin_categories import Command

# You have to inherit Command for this to be a
# command plugin:

class CommandServe(Command):
    """Start test server."""

    name = "serve"
    doc_usage = "[options]"
    doc_purpose = "start the test webserver"

    cmd_options = (
        {
            'name': 'port',
            'short': 'p',
            'long': 'port',
            'default': 8000,
            'type': int,
            'help': 'Port number (default: 8000)',
        },
        {
            'name': 'address',
            'short': 'a',
            'long': '--address',
            'type': str,
            'default': '127.0.0.1',
            'help': 'Address to bind (default: 127.0.0.1)',
        },
    )

    def _execute(self, options, args):
        """Start test server."""
        out_dir = self.site.config['OUTPUT_FOLDER']
        if not os.path.isdir(out_dir):
            print("Error: Missing '{0}' folder?".format(out_dir))
        else:
            os.chdir(out_dir)
            httpd = HTTPServer((options['address'], options['port']),
                               OurHTTPRequestHandler)
            sa = httpd.socket.getsockname()
            print("Serving HTTP on", sa[0], "port", sa[1], "...")
```

```
            httpd.serve_forever()
```

As mentioned above, a plugin can have options, which the user can see by doing `nikola help command` and can later use, for example:

```
$ nikola help serve
Purpose: start the test webserver
Usage:   nikola serve [options]

Options:
-p ARG, --port=ARG        Port number (default: 8000)
-a ARG, ----address=ARG   Address to bind (default: 127.0.0.1)

$ nikola serve -p 9000
Serving HTTP on 127.0.0.1 port 9000 ...
```

So, what can you do with commands? Well, anything you want, really. I have implemented a sort of planet using it. So, be creative, and if you do something interesting, let me know ;-)

TemplateSystem Plugins

Nikola supports Mako and Jinja2. If you prefer some other templating system, then you will have to write a `TemplateSystem` plugin. Here's how they work. First, you have to create a `.plugin` file. Here's the one for the Mako plugin:

```
[Core]
Name = mako
Module = mako

[Documentation]
Author = Roberto Alsina
Version = 0.1
Website = https://getnikola.com
Description = Support for Mako templates.
```

Note: If you want to publish your plugin on the Plugin Index, read the docs for the Index (and the .plugin file examples and explanations).

You will have to replace "mako" with your template system's name, and other data in the obvious ways.

The "Module" option is the name of the module, which has to look something like this, a stub for a hypothetical system called "Templater":

```
from nikola.plugin_categories import TemplateSystem

# You have to inherit TemplateSystem

class TemplaterTemplates(TemplateSystem):
    """Wrapper for Templater templates."""

    # name has to match Name in the .plugin file
    name = "templater"

    # A list of directories where the templates will be
```

```
    # located. Most template systems have some sort of
    # template loading tool that can use this.
    def set_directories(self, directories, cache_folder):
        """Sets the list of folders where templates are located and cache."""
        pass

    # You *must* implement this, even if to return []
    # It should return a list of all the files that,
    # when changed, may affect the template's output.
    # usually this involves template inheritance and
    # inclusion.
    def template_deps(self, template_name):
        """Returns filenames which are dependencies for a template."""
        return []

    def render_template(self, template_name, output_name, context):
        """Renders template to a file using context.

        This must save the data to output_name *and* return it
        so that the caller may do additional processing.
        """
        pass

    # The method that does the actual rendering.
    # template_name is the name of the template file,
    # context is a dictionary containing the data the template
    # uses for rendering.
    def render_template_to_string(self, template, context):
        """Renders template to a string using context. """
        pass

    def inject_directory(self, directory):
        """Injects the directory with the lowest priority in the
        template search mechanism."""
        pass
```

You can see a real example in the Jinja plugin

Task Plugins

If you want to do something that depends on the data in your site, you probably want to do a `Task` plugin, which will make it be part of the `nikola build` command. These are the currently available tasks, all provided by plugins:

Other Tasks

There are also `LateTask` plugins, which are executed later, and `TaskMultiplier` plugins that take a task and create more tasks out of it.

```
$ nikola list
Scanning posts....done!
build_bundles
build_less
copy_assets
copy_files
```

```
post_render
redirect
render_archive
render_galleries
render_galleries_clean
render_indexes
render_listings
render_pages
render_posts
render_rss
render_site
render_sources
render_tags
sitemap
```

These have access to the `site` object which contains your timeline and your configuration.

The critical bit of Task plugins is their `gen_tasks` method, which `yields` doit tasks.

The details of how to handle dependencies, etc., are a bit too much for this document, so I'll just leave you with an example, the `copy_assets` task. First the `task_copy_assets.plugin` file, which you should copy and edit in the logical ways:

```
[Core]
Name = copy_assets
Module = task_copy_assets

[Documentation]
Author = Roberto Alsina
Version = 0.1
Website = https://getnikola.com
Description = Copy theme assets into output.
```

Note: If you want to publish your plugin on the Plugin Index, read the docs for the Index (and the .plugin file examples and explanations).

And the `task_copy_assets.py` file, in its entirety:

```
import os

from nikola.plugin_categories import Task
from nikola import utils

# Have to inherit Task to be a task plugin
class CopyAssets(Task):
    """Copy theme assets into output."""

    name = "copy_assets"

    # This yields the tasks
    def gen_tasks(self):
        """Create tasks to copy the assets of the whole theme chain.

        If a file is present on two themes, use the version
        from the "youngest" theme.
        """
```

```
        # I put all the configurations and data the plugin uses
        # in a dictionary because utils.config_changed will
        # make it so that if these change, this task will be
        # marked out of date, and run again.

        kw = {
            "themes": self.site.THEMES,
            "output_folder": self.site.config['OUTPUT_FOLDER'],
            "filters": self.site.config['FILTERS'],
        }

        tasks = {}
        for theme_name in kw['themes']:
            src = os.path.join(utils.get_theme_path(theme_name), 'assets')
            dst = os.path.join(kw['output_folder'], 'assets')
            for task in utils.copy_tree(src, dst):
                if task['name'] in tasks:
                    continue
                tasks[task['name']] = task
                task['uptodate'] = task.get('uptodate', []) + \
                    [utils.config_changed(kw)]
                task['basename'] = self.name
                # If your task generates files, please do this.
                yield utils.apply_filters(task, kw['filters'])
```

PageCompiler Plugins

These plugins implement markup languages, they take sources for posts or pages and create HTML or other output files. A good example is the misaka plugin or the built-in compiler plugins.

They must provide:

`compile` Function that builds a file.

`create_post` Function that creates an empty file with some metadata in it.

If the compiler produces something other than HTML files, it should also implement `extension` which returns the preferred extension for the output file.

These plugins can also be used to extract metadata from a file. To do so, the plugin must set `supports_metadata` to `True` and implement `read_metadata` that will return a dict containing the metadata contained in the file. Optionally, it may list `metadata_conditions` (see *MetadataExtractor Plugins* below)

MetadataExtractor Plugins

Plugins that extract metadata from posts. If they are based on post content, they must implement `_extract_metadata_from_text` (takes source of a post returns a dict of metadata). They may also implement `split_metadata_from_text`, `extract_text`. If they are based on filenames, they only need `extract_filename`. If `support_write` is set to True, `write_metadata` must be implemented.

Every extractor must be configured properly. The `name`, `source` (from the `MetaSource` enum in `metadata_extractors`) and `priority` (`MetaPriority`) fields are mandatory. There might also be a list of `conditions` (tuples of `MetaCondition, arg`), used to check if an extractor can provide metadata, a compiled regular expression used to split metadata (`split_metadata_re`, may be `None`, used by default

`split_metadata_from_text`), a list of `requirements` (3-tuples: import name, pip name, friendly name), `map_from` (name of `METADATA_MAPPING` to use, if any) and `supports_write` (whether the extractor supports writing metadata in the desired format).

For more details, see the definition in `plugin_categories.py` and default extractors in `metadata_extractors.py`.

RestExtension Plugins

Implement directives for reStructuredText, see media.py for a simple example.

If your output depends on a config value, you need to make your post record a dependency on a pseudo-path, like this:

```
####MAGIC####CONFIG:OPTIONNAME
```

Then, whenever the `OPTIONNAME` option is changed in conf.py, the file will be rebuilt.

If your directive depends or may depend on the whole timeline (like the `post-list` directive, where adding new posts to the site could make it stale), you should record a dependency on the pseudo-path `####MAGIC####TIMELINE`.

MarkdownExtension Plugins

Implement Markdown extensions, see mdx_nikola.py for a simple example.

Note that Python markdown extensions are often also available as separate packages. This is only meant to ship extensions along with Nikola.

SignalHandler Plugins

These plugins extend the `SignalHandler` class and connect to one or more signals via blinker.

The easiest way to do this is to reimplement `set_site()` and just connect to whatever signals you want there.

Currently Nikola emits the following signals:

`sighandlers_loaded` Right after SignalHandler plugin activation.

`initialized` When all tasks are loaded.

`configured` When all the configuration file is processed. Note that plugins are activated before this is emitted.

`scanned` After posts are scanned.

`new_post` / `new_page` When a new post is created, using the `nikola new_post`/`nikola new_page` commands. The signal data contains the path of the file, and the metadata file (if there is one).

`existing_post` / `existing_page` When a new post fails to be created due to a title conflict. Contains the same data as `new_post`.

`deployed` When the `nikola deploy` command is run, and there is at least one new entry/post since `last_deploy`. The signal data is of the form:

```
{
 'last_deploy: # datetime object for the last deployed time,
 'new_deploy': # datetime object for the current deployed time,
 'clean': # whether there was a record of a last deployment,
 'deployed': # all files deployed after the last deploy,
 'undeployed': # all files not deployed since they are either future posts/drafts
}
```

compiled When a post/page is compiled from its source to html, before anything else is done with it. The signal data is in the form:

```
{
 'source': # the path to the source file
 'dest': # the path to the cache file for the post/page
 'post': # the Post object for the post/page
}
```

One example is the deploy_hooks plugin.

ConfigPlugin Plugins

Does nothing specific, can be used to modify the site object (and thus the config).

Put all the magic you want in `set_site()`, and don't forget to run the one from `super()`. Example plugin: navstories

PostScanner Plugins

Get posts and pages from "somewhere" to be added to the timeline. The only currently existing plugin of this kind reads them from disk.

CHAPTER 22

Plugin Index

There is a plugin index, which stores all of the plugins for Nikola people wanted to share with the world.

You may want to read the README for the Index if you want to publish your package there.

CHAPTER 23

Path/Link Resolution Mechanism

Any plugin can register a function using `Nikola.register_path_handler` to allow resolution of paths and links. These are useful for templates, which can access them via `_link`.

For example, you can always get a link to the path for the feed of the "foo" tag by using `_link('tag_rss', 'foo')` or the `link://tag_rss/foo` URL.

Here's the relevant code from the tag plugin.

```
# In set_site
site.register_path_handler('tag_rss', self.tag_rss_path)

# And these always take name and lang as arguments and returl a list of
# path elements.
def tag_rss_path(self, name, lang):
    return [_f for _f in [self.site.config['TRANSLATIONS'][lang],
                          self.site.config['TAG_PATH'], self.slugify_name(name, lang)␣
↪+ ".xml"] if
            _f]
```

CHAPTER 24

Template Hooks

Plugins can use a hook system for adding stuff into templates. In order to use it, a plugin must register itself. The following hooks currently exist:

- `extra_head` (not equal to the config option!)
- `body_end` (not equal to the config option!)
- `page_header`
- `menu`
- `menu_alt` (right-side menu in bootstrap, after `menu` in base)
- `page_footer`

For example, in order to register a script into `extra_head`:

```
# In set_site
site.template_hooks['extra_head'].append('<script src="/assets/js/fancyplugin.js">')
```

There is also another API available. It allows use of dynamically generated HTML:

```
# In set_site
def generate_html_bit(name, ftype='js'):
    """Generate HTML for an asset."""
    return '<script src="/assets/{t}/{n}.{t}">'.format(n=name, t=ftype)

site.template_hooks['extra_head'].append(generate_html_bit, False, 'fancyplugin',␣
↪ftype='js')
```

The second argument to `append()` is used to determine whether the function needs access to the current template context and the site. If it is set to `True`, the function will also receive `site` and `context` keyword arguments. Example use:

```
# In set_site
def greeting(addr, endswith='', site=None, context=None):
    """Greet someone."""
```

```
    if context['lang'] == 'en':
        greet = u'Hello'
    elif context['lang'] == 'es':
        greet = u'¡Hola'

    t = u' BLOG_TITLE = {0}'.format(site.config['BLOG_TITLE'](context['lang']))

    return u'<h3>{greet} {addr}{endswith}</h3>'.format(greet=greet, addr=addr,
    endswith=endswith) + t

site.template_hooks['page_header'].append(greeting, True, u'Nikola Tesla', endswith=u
↪'!')
```

Dependencies for template hooks:

- if the input is a string, the string value, alongside arguments to `append`, is used for calculating dependencies
- if the input is a callable, it attempts `input.template_registry_identifier`, then `input.__doc__`, and if neither is available, it uses a static string.

Make sure to provide at least a docstring, or a identifier, to ensure rebuilds work properly.

CHAPTER 25

Shortcodes

Some (hopefully all) markup compilers support shortcodes in these forms:

```
{{% raw %}}{{% foo %}}  # No arguments
{{% foo bar %}}  # One argument, containing "bar"
{{% foo bar baz=bat %}}  # Two arguments, one containing "bar", one called "baz"␣
↪containing "bat"

{{% foo %}}Some text{{% /foo %}}  # one argument called "data" containing "Some text"{
↪{% /raw %}}
```

So, if you are creating a plugin that generates markup, it may be a good idea to register it as a shortcode in addition of to restructured text directive or markdown extension, thus making it available to all markup formats.

To implement your own shortcodes from a plugin, you can create a plugin inheriting `ShortcodePlugin` and from its `set_site` method, call

`Nikola.register_shortcode(name, func)` with the following arguments:

`name`: Name of the shortcode ("foo" in the examples above)

`func`: A function that will handle the shortcode

The shortcode handler **must** return a two-element tuple, `(output, dependencies)`

`output`: The text that will replace the shortcode in the document.

`dependencies`: A list of all the files on disk which will make the output be considered out of date. For example, if the shortcode uses a template, it should be the path to the template file.

The shortcode handler **must** accept the following named arguments (or variable keyword arguments):

`site`: An instance of the Nikola class, to access site state

`data`: If the shortcut is used as opening/closing tags, it will be the text between them, otherwise `None`.

`lang`: The current language.

If the shortcode tag has arguments of the form `foo=bar` they will be passed as named arguments. Everything else will be passed as positional arguments in the function call.

So, for example:

```
{{% raw %}}{{% foo bar baz=bat beep %}}Some text{{% /foo %}}{{% /raw %}}
```

Assuming you registered `foo_handler` as the handler function for the shortcode named `foo`, this will result in the following call when the above shortcode is encountered:

```
foo_handler("bar", "beep", baz="bat", data="Some text", site=whatever)
```

Template-based Shortcodes

Another way to define a new shortcode is to add a template file to the `shortcodes` directory of your site. The template file must have the shortcode name as the basename and the extension `.tmpl`. For example, if you want to add a new shortcode named `foo`, create the template file as `shortcodes/foo.tmpl`.

When the shortcode is encountered, the matching template will be rendered with its context provided by the arguments given in the shortcode. Keyword arguments are passed directly, i.e. the key becomes the variable name in the template namespace with a matching string value. Non-keyword arguments are passed as string values in a tuple named `_args`. As for normal shortcodes with a handler function, `site` and `data` will be added to the keyword arguments.

Example:

The following shortcode:

```
{{% raw %}}{{% foo bar="baz" spam %}}{{% /raw %}}
```

With a template in `shortcodes/foo.tmpl` with this content (using Jinja2 syntax in this example)

```
<div class="{{ _args[0] if _args else 'ham' }}">{{ bar }}</div>
```

Will result in this output

```
<div class="spam">baz</div>
```

CHAPTER 26

State and Cache

Sometimes your plugins will need to cache things to speed up further actions. Here are the conventions for that:

- If it's a file, put it somewhere in `self.site.config['CACHE_FOLDER']` (defaults to `cache/`.
- If it's a value, use `self.site.cache.set(key, value)` to set it and `self.site.cache.get(key)` to get it. The key should be a string, the value should be json-encodable (so, be careful with datetime objects)

The values and files you store there can **and will** be deleted sometimes by the user. They should always be things you can reconstruct without lossage. They are throwaways.

On the other hand, sometimes you want to save something that is **not** a throwaway. These are things that may change the output, so the user should not delete them. We call that **state**. To save state:

- If it's a file, put it somewhere in the working directory. Try not to do that please.
- If it's a value, use `self.site.state.set(key, value)` to set it and `self.state.cache.get(key)` to get it. The key should be a string, the value should be json-encodable (so, be careful with datetime objects)

The `cache` and `state` objects are rather simplistic, and that's intentional. They have no default values: if the key is not there, you will get `None` and like it. They are meant to be both threadsafe, but hey, who can guarantee that sort of thing?

There are no sections, and no access protection, so let's not use it to store passwords and such. Use responsibly.

CHAPTER 27

Nikola Internals

When trying to guide someone into adding a feature in Nikola, it hit me that while the way it's structured makes sense **to me** it is far from obvious.

So, this is a short document explaining what each piece of Nikola does and how it all fits together.

Nikola is a Pile of Plugins Most of Nikola is implemented as plugins using Yapsy. You can ignore that they are plugins and just think of them as regular python modules and packages with a funny little `.plugin` file next to them.

So, 90% of the time, what you want to do is either write a new plugin or extend an existing one.

There are several kinds of plugins, all implementing interfaces defined in `nikola/plugin_categories.py` and documented in Extending Nikola

If your plugin has a dependency, please make sure it doesn't make Nikola throw an exception when the dependency is missing. Try to fail gracefully with an informative message.

Commands are plugins When you use `nikola foo` you are using the plugin `command/foo`. Those are used to extend Nikola's command line. Their interface is defined in the `Command` class. They take options and arguments and do whatever you want, so go wild.

The `build` command is special The `build` command triggers a whole lot of things, and is the core of Nikola because it's the one that you use to build sites. So it deserves its own section.

The Build Command

Nikola's goal is similar, deep at heart, to a Makefile. Take sources, compile them into something, in this case a website. Instead of a Makefile, Nikola uses doit

Doit has the concept of "tasks". The 1 minute summary of tasks is that they have:

actions What the task **does**. For example, convert a markdown document into HTML.

dependencies If this file changes, then we need to redo the actions. If this configuration option changes, redo it, etc.

targets Files that the action generates. No two actions can have the same targets.

basename:name Each task is identified by either a name or a basename:name pair.

More about tasks

If you ever want to do your own tasks, you really should read the doit documentation on tasks

So, what Nikola does, when you use the build command, is to read the configuration `conf.py` from the current folder, instantiate the `Nikola` class, and have it generate a whole list of tasks for doit to process. Then doit will decide which tasks need doing, and do them, in the right order.

The place where the tasks are generated is in `Nikola.gen_tasks`, which collects tasks from all the plugins inheriting `BaseTask`, massages them a bit, then passes them to doit.

So, if you want things to happen on `build` you want to create a Task plugin, or extend one of the existing ones.

Tests

While Nikola is not a hardcore TDD project, we like tests. So, please add them if you can. You can write unit tests or integration tests. (Doctests are not supported anymore due to fragility.)

Posts and Pages

Nikola has a concept of posts and pages. Both are more or less the same thing, except posts are added into RSS feeds and pages are not. All of them are in a list called "the timeline" formed by objects of class `Post`.

When you are creating a task that needs the list of posts and/or pages (for example, the RSS creation plugin) on task execution time, your plugin should call `self.site.scan_posts()` in `gen_tasks` to ensure the timeline is created and available in `self.site.timeline`. You should not modify the timeline, because it will cause consistency issues.

scan_posts

The `Nikola.scan_posts` function can be used in plugins to force the timeline creation, for example, while creating the tasks.

Your plugin can use the timeline to generate "stuff" (technical term). For example, Nikola comes with plugins that use the timeline to create a website (surprised?).

The workflow included with nikola is as follows (incomplete!):

1. The post is assigned a compiler based on its extension and the `COMPILERS` option.
2. The compiler is applied to the post data and a "HTML fragment" is produced. That fragment is stored in a cache (the `posts` plugin).
3. The configured theme has templates (and a template engine), which are applied to the post's HTML fragment and metadata (the `pages` plugin).
4. The original sources for the post are copied to some accessible place (the `sources` plugin).
5. If the post is tagged, some pages and RSS feeds for each tag are updated (the `tags` plugin).
6. If the post is new, it's included in the blog's RSS feed (the `rss` plugin).

7. The post is added in the right place in the index pages for the blog (the `indexes` plugin).
8. CSS/JS/Images for the theme are put in the right places (the `copy_assets` and `bundles` plugins).
9. A File describing the whole site is created (the `sitemap` plugin).

You can add whatever you want to that list: just create a plugin for it.

You can also expand Nikola's capabilities at several points:

compilers Nikola supports a variety of markups. If you want to add another one, you need to create a `Compiler` plugin.

templates Nikola's themes can use Jinja2 or Mako templates. If you prefer another template system, you have to create a `TemplateSystem` plugin.

themes To change how the generated site looks, you can create custom themes.

And of course, you can also replace or extend each of the existing plugins.

CHAPTER 28

Nikola Architecture

CHAPTER 29

Using Alternative Social Buttons with Nikola

Version 7.8.8

Contents

The Default

By Default, the themes provided with Nikola will add to your pages a "slide in" widget at the bottom right of the page, provided by Addthis. This is the HTML code for that:

```
<!-- Social buttons -->
<div id="addthisbox" class="addthis_toolbox addthis_peekaboo_style
    addthis_default_style addthis_label_style addthis_32x32_style">
<a class="addthis_button_more">Share</a>
<ul><li><a class="addthis_button_facebook"></a>
<li><a class="addthis_button_google_plusone_share"></a>
<li><a class="addthis_button_linkedin"></a>
<li><a class="addthis_button_twitter"></a>
</ul>
</div>
<script src="//s7.addthis.com/js/300/addthis_widget.js#pubid=ra-4f7088a56bb93798"></
↪script>
```

```
<!-- End of social buttons -->
"""
```

You can change that using the SOCIAL_BUTTONS_CODE option in your conf.py. In some cases, just doing that will be enough but in others, it won't. This document tries to describe all the bits involved in making this work correctly.

Part 1: SOCIAL_BUTTONS_CODE Social sharing services like addthis and others will provide you a HTML snippet. If it is self-contained, then just setting SOCIAL_BUTTONS_CODE may be enough. Try :-)

Part 2: The theme The SOCIAL_BUTTONS_CODE HTML fragment will be embedded *somewhere* by the theme. Whether that is the correct place or not is not something the theme author can truly know, so it is possible that you may have to tweak the base.html template to make it look good.

Part 3: BODY_END and EXTRA_HEAD_DATA Some social sharing code requires JS execution that depends on JQuery being available (example: SocialSharePrivacy). It's good practice (and often, the only way that will work) to put those at the end of <BODY>, and one easy way to do that is to put them in BODY_END

On the other hand, it's possible that it requires you to load some CSS files. The right place for that is in the document's <HEAD> so they should be added in EXTRA_HEAD_DATA

Part 4: assets For sharing code that doesn't rely on a social sharing service, you may need to add CSS, Image, or JS files to your site

ShareNice

Sharenice is "written in order to provide social sharing features to web developers and website administrators who wish to maintain and protect their users' privacy" which sounds cool to me.

Let's go step by step into integrating the hosted version of ShareNice into a Nikola site.

For testing purposes, let's do it on a demo site:

```
$ nikola init --demo sharenice_test
A new site with example data has been created at sharenice_test.
See README.txt in that folder for more information.
$ cd sharenice_test/
```

To see what's going on, let's start Nikola in "auto mode". This should build the site and open a web browser showing the default configuration, with the AddThis widget:

```
$ nikola auto -b
```

First, let's add the HTML snippet that will show the sharing options. In your conf.py, set this, which is the HTML code suggested by ShareNice:

```
SOCIAL_BUTTONS_CODE = """<div id="shareNice" data-share-label="Share"
    data-color-scheme="black" data-icon-size="32" data-panel-bottom="plain"
    data-services="plus.google.com,facebook.com,digg.com,email,delicious.com,twitter.
↪com"
    style="float:right"></div>"""

BODY_END = """<script src="http://graingert.co.uk/shareNice/code.js"></script>"""
```

And you should now see a sharing box at the bottom right of the page.

Main problem remaining is that it doesn't really look good and integrated in the page layout. I suggest changing the code to this which looks nicer, but still has some placement issues:

```
SOCIAL_BUTTONS_CODE = """<div id="shareNice" data-share-label="Share"
    data-color-scheme="black" data-icon-size="32" data-panel-bottom="plain"
    data-services="plus.google.com,facebook.com,email,twitter.com"
    style="position: absolute; left: 20px; top: 60px;"></div>"""
```

If anyone comes up with a better idea of styling/placement, just let me know ;-)

One bad bit of this so far is that you are now using a script from another site, and that doesn't let Nikola perform as many optimizations to your page as it could. So, if you really want to go the extra mile to save a few KB and round trips, you *could* install your own copy from the github repo and use that instead of the copy at sharenice.org.

Then, you can create your own theme inheriting from the one you are using and add the CSS and JS files from ShareNice into your `bundles` configuration so they are combined and minified.

SocialSharePrivacy

The Hard Way

SocialSharePrivacy is "a jQuery plugin that lets you add social share buttons to your website that don't allow the social sites to track your users." Nice!

Let's go step-by-step into integrating SocialSharePrivacy into a Nikola site. To improve privacy, they recommend you not use the hosted service so we'll do it the hard way, by getting and distributing everything in our own site.

https://github.com/panzi/SocialSharePrivacy

For testing purposes, let's do it on a demo site:

```
$ nikola init --demo ssp_test
A new site with example data has been created at ssp_test.
See README.txt in that folder for more information.
$ cd ssp_test/
```

To see what's going on, let's start Nikola in "auto mode". This should build the site and open a web browser showing the default configuration, with the AddThis widget:

```
$ nikola auto -b
```

Now, download the current version and unzip it. You will have a `SocialSharePrivacy-master` folder with lots of stuff in it.

First, we need to build it (this requires a working and modern uglifyjs, this may not be easy):

```
$ cd SocialSharePrivacy-master
$ sh build.sh -m gplus,twitter,facebook,mail -s "/assets/css/socialshareprivacy.css" -
↪a off
```

You will now have several files in a `build` folder. We need to bring them into the site:

```
$ cp -Rv SocialSharePrivacy-master/build/* files/
$ cp -R SocialSharePrivacy-master/images/ files/assets/
```

Edit your `conf.py`:

```
BODY_END = """
<script src="/javascripts/jquery.socialshareprivacy.min.js"></script>
<script>
```

```
$(document).ready(function () {
    $('.share').socialSharePrivacy();
});
</script>
"""

SOCIAL_BUTTONS_CODE = """<div class="share"></div>"""
```

In my experience this produces a broken, duplicate, semi-working thing. YMMV and if you make it work correctly, let me know how :-)

The Easy Way

Go to http://panzi.github.io/SocialSharePrivacy/ and use the provided form to get the code. Make sure you check "I already use JQuery" if you are using one of the themes that require it, like site or default, select the services you want, and use your disqus name if you have one.

It will give you 3 code snippets:

"Insert this once in the head of your page" Put it in `BODY_END`

"Insert this wherever you want a share widget displayed" Put it in `SOCIAL_BUTTONS_CODE`

"Insert this once anywhere after the other code" Put it in `BODY_END`

That should give you a working integration (not tested)

Nikola supports special links with the syntax `link://kind/name`. In the templates you can also use `_link(kind, name)`. You can also add query strings (`?key=value`) for extra arguments, or pass keyword arguments to `_link` in templates (support and behavior depends on path handlers themselves)

Here are the descriptions for all the supported kinds.

archive Link to archive path, name is the year.

Example:

link://archive/2013 => /archives/2013/index.html

archive_atom Link to archive Atom path, name is the year (archive pages must be indexes).

Example:

link://archive_atom/2013 => /archives/2013/index.atom

author Link to an author's page.

Example:

link://author/joe => /authors/joe.html

author_atom Link to an author's Atom feed.

Example:

link://author_atom/joe => /authors/joe.atom

author_index Link to the authors index.

Example:

link://authors/ => /authors/index.html

author_rss Link to an author's RSS feed.

Example:

link://author_rss/joe => /authors/joe.xml

category A link to a category. Takes page number as optional keyword argument.

Example:

link://category/dogs => /categories/dogs.html

category_atom A link to a category's Atom feed.

Example:

link://category_atom/dogs => /categories/dogs.atom

category_index A link to the category index.

Example:

link://category_index => /categories/index.html

category_rss A link to a category's RSS feed.

Example:

link://category_rss/dogs => /categories/dogs.xml

filename Link to post or page by source filename.

Example:

link://filename/manual.txt => /docs/handbook.html

gallery Link to an image gallery's path.

It will try to find a gallery with that name if it's not ambiguous or with that path. For example:

link://gallery/london => /galleries/trips/london/index.html

link://gallery/trips/london => /galleries/trips/london/index.html

gallery_global Link to the global gallery path, which contains all the images in galleries.

There is only one copy of an image on multilingual blogs, in the site root.

link://gallery_global/london => /galleries/trips/london/index.html

link://gallery_global/trips/london => /galleries/trips/london/index.html

(a `gallery` link could lead to eg. /en/galleries/trips/london/index.html)

gallery_rss Link to an image gallery's RSS feed.

It will try to find a gallery with that name if it's not ambiguous or with that path. For example:

link://gallery_rss/london => /galleries/trips/london/rss.xml

link://gallery_rss/trips/london => /galleries/trips/london/rss.xml

index Link to a numbered index.

Example:

link://index/3 => /index-3.html

index_atom Link to a numbered Atom index.

Example:

link://index_atom/3 => /index-3.atom

index_rss A link to the RSS feed path.

Example:

link://rss => /blog/rss.xml

listing Return a link to a listing.

It will try to use the file name if it's not ambiguous, or the file path.

Example:

link://listing/hello.py => /listings/tutorial/hello.py.html

link://listing/tutorial/hello.py => /listings/tutorial/hello.py.html

listing_source Return a link to the source code for a listing.

It will try to use the file name if it's not ambiguous, or the file path.

Example:

link://listing_source/hello.py => /listings/tutorial/hello.py

link://listing_source/tutorial/hello.py => /listings/tutorial/hello.py

post_path Link to the destination of an element in the POSTS/PAGES settings.

Example:

link://post_path/posts => /blog

root Link to the current language's root.

Example:

link://root_path => /

link://root_path => /translations/spanish/

rss A link to the RSS feed path.

Example:

link://rss => /blog/rss.xml

section_index Link to the index for a section.

Example:

link://section_index/cars => /cars/index.html

section_index_atom Link to the Atom index for a section.

Example:

link://section_index_atom/cars => /cars/index.atom

section_index_rss Link to the RSS feed for a section.

Example:

link://section_index_rss/cars => /cars/rss.xml

slug Return a link to a post with given slug, if not ambiguous.

Example:

link://slug/yellow-camaro => /posts/cars/awful/yellow-camaro/index.html

tag A link to a tag's page. Takes page number as optional keyword argument.

Example:

link://tag/cats => /tags/cats.html

tag_atom A link to a tag's Atom feed.

Example:

link://tag_atom/cats => /tags/cats.atom

tag_index A link to the tag index.

Example:

link://tag_index => /tags/index.html

tag_rss A link to a tag's RSS feed.

Example:

link://tag_rss/cats => /tags/cats.xml

CHAPTER 30

nikola

nikola package

Subpackages

nikola.packages package

Subpackages

nikola.packages.datecond package

Module contents

Date range parser.

nikola.packages.datecond.**date_in_range**(*date_range*, *date*, *debug=False*, *now=None*)

Check if date is in the range specified.

Format: * comma-separated clauses (AND) * clause: attribute comparison_operator value (spaces optional)

- attribute: year, month, day, hour, month, second, weekday, isoweekday or empty for full datetime
- comparison_operator: == != <= >= < >
- value: integer, 'now' or dateutil-compatible date input

The optional *now* parameter can be used to provide a specific *now* value (if none is provided, datetime.datetime.now() is used).

nikola.packages.tzlocal package

Submodules

nikola.packages.tzlocal.darwin module

tzlocal for OS X.

nikola.packages.tzlocal.darwin.**get_localzone**()
Get the computers configured local timezone, if any.

nikola.packages.tzlocal.darwin.**reload_localzone**()
Reload the cached localzone. You need to call this if the timezone has changed.

nikola.packages.tzlocal.unix module

tzlocal for UNIX.

nikola.packages.tzlocal.unix.**get_localzone**()
Get the computers configured local timezone, if any.

nikola.packages.tzlocal.unix.**reload_localzone**()
Reload the cached localzone. You need to call this if the timezone has changed.

nikola.packages.tzlocal.win32 module

tzlocal for Windows.

nikola.packages.tzlocal.win32.**get_localzone**()
Return the zoneinfo-based tzinfo object that matches the Windows-configured timezone.

nikola.packages.tzlocal.win32.**get_localzone_name**()
Get local time zone name.

nikola.packages.tzlocal.win32.**reload_localzone**()
Reload the cached localzone. You need to call this if the timezone has changed.

nikola.packages.tzlocal.win32.**valuestodict**(*key*)
Convert a registry key's values to a dictionary.

nikola.packages.tzlocal.windows_tz module

Windows timezone names.

Module contents

tzlocal init.

Module contents

Third-party packages for Nikola.

nikola.plugins package

Subpackages

nikola.plugins.command package

Subpackages

nikola.plugins.command.auto package

Module contents

Automatic rebuilds for Nikola.

class nikola.plugins.command.auto.**CommandAuto**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Automatic rebuilds for Nikola.

cmd_options = [{'help': 'Port number (default: 8000)', 'long': 'port', 'name': 'port', 'type': <class 'int'>, 'short': 'p',

dns_sd = None

do_rebuild(*event*)
Rebuild the site.

do_refresh(*event*)
Refresh the page.

doc_purpose = 'builds and serves a site; automatically detects site changes, rebuilds, and optionally refreshes a browser

file_filter(*mimetype*, *data*)
Apply necessary changes to document before serving.

has_server = True

inject_js(*data*)
Inject livereload.js.

logger = None

name = 'auto'

remove_base_tag(*data*)
Comment out any <base> to allow local resolution of relative URLs.

serve_static(*environ*, *start_response*)
Trivial static file server.

class nikola.plugins.command.auto.**ConfigWatchHandler**(*configuration_filename*, *function*)
Bases: **:class:'watchdog.events.FileSystemEventHandler'**

A Nikola-specific handler for Watchdog that handles the config file (as a workaround).

on_any_event(*event*)
Call the provided function on any event.

class nikola.plugins.command.auto.**LRSocket**(**a*, ***kw*)
Bases: **:class:'ws4py.websocket.WebSocket'**

Speak Livereload protocol.

notify(*sender*, *path*)
Send reload requests to the client.

received_message(*message*)
Handle received message.

send_error(*sender*, *error=None*)
Send reload requests to the client.

class nikola.plugins.command.auto.**OurWatchHandler**(*function*)
Bases: **:class:'watchdog.events.FileSystemEventHandler'**

A Nikola-specific handler for Watchdog.

on_any_event(*event*)
Call the provided function on any event.

nikola.plugins.command.auto.**finish_response**(*self*)
Monkeypatched finish_response that ignores broken pipes.

nikola.plugins.command.rst2html package

Module contents

Compile reStructuredText to HTML, using Nikola architecture.

class nikola.plugins.command.rst2html.**CommandRst2Html**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Compile reStructuredText to HTML, using Nikola architecture.

doc_purpose = 'compile reStructuredText to HTML files'

doc_usage = 'infile'

name = 'rst2html'

needs_config = False

Submodules

nikola.plugins.command.bootswatch_theme module

Given a swatch name from bootswatch.com and a parent theme, creates a custom theme.

class nikola.plugins.command.bootswatch_theme.**CommandBootswatchTheme**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Given a swatch name from bootswatch.com and a parent theme, creates a custom theme.

cmd_options = [{'help': 'New theme name (default: custom)', 'long': 'name', 'name': 'name', 'type': <class 'str'>, 'sh

doc_purpose = 'given a swatch name from bootswatch.com and a parent theme, creates a custom theme'

doc_usage = '[options]'

name = 'bootswatch_theme'

nikola.plugins.command.check module

Check the generated site.

class nikola.plugins.command.check.**CommandCheck**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Check the generated site.

analyze(*fname*, *find_sources=False*, *check_remote=False*)
Analyze links on a page.

cache = {}

checked_remote_targets = {}

clean_files()
Remove orphaned files.

cmd_options = [{'help': 'Check for dangling links', 'long': 'check-links', 'name': 'links', 'type': <class 'bool'>, 'short'

doc_purpose = 'check links and files in the generated site'

doc_usage = '[-v] (-l [–find-sources] [-r] | -f [–clean-files])'

existing_targets = set()

logger = None

name = 'check'

scan_files()
Check files in the site, find missing and orphaned files.

scan_links(*find_sources=False*, *check_remote=False*)
Check links on the site.

nikola.plugins.command.check.**fs_relpath_from_url_path**(*url_path*)
Create a filesystem relative path from an URL path.

nikola.plugins.command.check.**real_scan_files**(*site*, *cache=None*)
Scan for files.

nikola.plugins.command.console module

Start debugging console.

class nikola.plugins.command.console.**CommandConsole**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Start debugging console.

bpython(*willful=True*)
Run a bpython shell.

cmd_options = [{'help': 'Use bpython', 'long': 'bpython', 'name': 'bpython', 'type': <class 'bool'>, 'short': 'b', 'defau

doc_description = 'The site engine is accessible as 'site', the config file as 'conf', and commands are available as 'com

doc_purpose = 'start an interactive Python console with access to your site'

header = 'Nikola v8.0.0.dev0 – {0} Console (conf = configuration file, site = site engine, commands = nikola commands)'

ipython(*willful=True*)
Run an IPython shell.

name = 'console'

plain (*willful=True*)
Run a plain Python shell.

shells = ['ipython', 'bpython', 'plain']

nikola.plugins.command.deploy module

Deploy site.

class nikola.plugins.command.deploy.**CommandDeploy**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Deploy site.

doc_description = 'Deploy the site by executing deploy commands from the presets listed on the command line. If no

doc_purpose = 'deploy the site'

doc_usage = '[preset [preset...]]'

logger = None

name = 'deploy'

nikola.plugins.command.github_deploy module

Deploy site to GitHub Pages.

class nikola.plugins.command.github_deploy.**CommandGitHubDeploy**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Deploy site to GitHub Pages.

cmd_options = [{'help': 'Commit message (default: Nikola auto commit.)', 'long': 'message', 'name': 'commit_messag

doc_description = 'This command can be used to deploy your site to GitHub Pages. It uses ghp-import to do this task

doc_purpose = 'deploy the site to GitHub Pages'

doc_usage = '[-m COMMIT_MESSAGE]'

logger = None

name = 'github_deploy'

nikola.plugins.command.github_deploy.**check_ghp_import_installed**()
Check if ghp-import is installed.

nikola.plugins.command.github_deploy.**uni_check_output**(**args*, ***kwargs*)
Run command and return output as Unicode (UTf-8).

nikola.plugins.command.import_wordpress module

Import a WordPress dump.

class nikola.plugins.command.import_wordpress.**CommandImportWordpress**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**, **:class:'nikola.plugins.basic_import.ImportMixin'**

Import a WordPress dump.

all_tags = set()

cmd_options = [{'long': 'output-folder', 'name': 'output_folder', 'short': 'o', 'default': 'new_site', 'help': 'Location to

code_re1 = re.compile('\\[code.* lang.*?="(.*?)?".*\\](.*?)\\[/code\\]', re.MULTILINE|re.DOTALL)

code_re2 = re.compile('\\[sourcecode.* lang.*?="(.*?)?".*\\](.*?)\\[/sourcecode\\]', re.MULTILINE|re.DOTALL)

code_re3 = re.compile('\\[code.*?\\](.*?)\\[/code\\]', re.MULTILINE|re.DOTALL)

code_re4 = re.compile('\\[sourcecode.*?\\](.*?)\\[/sourcecode\\]', re.MULTILINE|re.DOTALL)

doc_purpose = 'import a WordPress dump'

doc_usage = '[options] wordpress_export_file'

download_url_content_to_file(*url*, *dst_path*)
Download some content (attachments) to a file.

classmethod get_channel_from_file(*filename*)
Get channel from XML file.

import_attachment(*item*, *wordpress_namespace*)
Import an attachment to the site.

import_postpage_item(*item*, *wordpress_namespace*, *out_folder=None*, *attachments=None*)
Take an item from the feed and creates a post file.

import_posts(*channel*)
Import posts into the site.

name = 'import_wordpress'

needs_config = False

populate_context(*channel*)
Populate context with config for the site.

process_item_if_attachment(*item*)
Process attachments.

process_item_if_post_or_page(*item*)
Process posts and pages.

classmethod read_xml_file(*filename*)
Read XML file into memory.

static transform_caption(*content*, *use_html=False*)
Transform captions.

transform_code(*content*)
Transform code blocks.

transform_content(*content*, *post_format*, *attachments*)
Transform content into appropriate format.

transform_multiple_newlines(*content*)
Replace multiple newlines with only two.

write_attachments_info(*path*, *attachments*)
Write attachments info file.

nikola.plugins.command.import_wordpress.**get_text_tag**(*tag*, *name*, *default*)
Get the text of an XML tag.

nikola.plugins.command.import_wordpress.**install_plugin**(*site*, *plugin_name*, *output_dir=None*, *show_install_notes=False*)
Install a Nikola plugin.

nikola.plugins.command.import_wordpress.**separate_qtranslate_content**(*text*)
Parse the content of a wordpress post or page and separate qtranslate languages.

qtranslate tags: <!–:LL–>blabla<!–:–>

nikola.plugins.command.init module

Create a new site.

class nikola.plugins.command.init.**CommandInit**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Create a new site.

static ask_questions(*target*, *demo=False*)
Ask some questions about Nikola.

cmd_options = [{'help': 'Do not ask questions about config.', 'long': 'quiet', 'name': 'quiet', 'type': <class 'bool'>, 'sho

classmethod copy_sample_site(*target*)
Copy sample site data to target directory.

static create_configuration(*target*)
Create configuration file.

static create_configuration_to_string()
Return configuration file as a string.

classmethod create_empty_site(*target*)
Create an empty site with directories only.

doc_purpose = 'create a Nikola site in the specified folder'

doc_usage = '[–demo] [–quiet] folder'

name = 'init'

needs_config = False

nikola.plugins.command.init.**format_default_translations_config**(*additional_languages*)
Adapt TRANSLATIONS setting for all additional languages.

nikola.plugins.command.init.**format_navigation_links**(*additional_languages*, *default_lang*, *messages*, *strip_indexes=False*)
Return the string to configure NAVIGATION_LINKS.

nikola.plugins.command.init.**prepare_config**(*config*)
Parse sample config with JSON.

nikola.plugins.command.init.**test_destination**(*destination*, *demo=False*)
Check if the destination already exists, which can break demo site creation.

nikola.plugins.command.install_theme module

nikola.plugins.command.new_page module

Create a new page.

class nikola.plugins.command.new_page.**CommandNewPage**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Create a new page.

cmd_options = [{'help': 'Title for the page.', 'long': 'title', 'name': 'title', 'type': <class 'str'>, 'short': 't', 'default': ''

doc_purpose = 'create a new page in the site'

doc_usage = '[options] [path]'

name = 'new_page'

nikola.plugins.command.new_post module

Create a new post.

class nikola.plugins.command.new_post.**CommandNewPost**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Create a new post.

cmd_options = [{'help': 'Create a page instead of a blog post. (see also: 'nikola new_page')', 'long': 'page', 'name': 'is

doc_purpose = 'create a new blog post or site page'

doc_usage = '[options] [path]'

filter_post_pages(*compiler*, *is_post*)
Return the correct entry from post_pages.

Information based on: * selected compilers * available compilers * post/page status

name = 'new_post'

print_compilers()
List all available compilers in a human-friendly format.

nikola.plugins.command.new_post.**get_date**(*schedule=False*, *rule=None*, *last_date=None*, *tz=None*, *iso8601=False*)
Return a date stamp, given a recurrence rule.

schedule - bool: whether to use the recurrence rule or not

rule - str: an iCal RRULE string that specifies the rule for scheduling posts

last_date - datetime: timestamp of the last post

tz - tzinfo: the timezone used for getting the current time.

iso8601 - bool: whether to force ISO 8601 dates (instead of locale-specific ones)

nikola.plugins.command.new_post.**get_default_compiler**(*is_post*, *compilers*, *post_pages*)
Given compilers and post_pages, return a reasonable default compiler for this kind of post/page.

nikola.plugins.command.orphans module

List all orphans.

class nikola.plugins.command.orphans.**CommandOrphans**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

List all orphans.

doc_description = 'List all orphans, i.e. all files that are in the output directory,\nbut are not generated by Nikola.\n

doc_purpose = 'list all orphans'

name = 'orphans'

nikola.plugins.command.plugin module

Manage plugins.

class nikola.plugins.command.plugin.**CommandPlugin**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Manage plugins.

cmd_options = [{'help': 'Install a plugin.', 'long': 'install', 'name': 'install', 'type': <class 'str'>, 'short': 'i', 'default':

do_install(*url*, *name*, *show_install_notes=True*)
Download and install a plugin.

do_uninstall(*name*)
Uninstall a plugin.

do_upgrade(*url*)
Upgrade all installed plugins.

doc_purpose = 'manage plugins'

doc_usage = '[-u url] [–user] [-i name] [-r name] [–upgrade] [-l] [–list-installed]'

get_json(*url*)
Download the JSON file with all plugins.

json = None

list_available(*url*)
List all available plugins.

list_installed()
List installed plugins.

name = 'plugin'

needs_config = False

output_dir = None

nikola.plugins.command.serve module

Start test server.

class nikola.plugins.command.serve.**CommandServe**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Start test server.

cmd_options = ({'help': 'Port number (default: 8000)', 'long': 'port', 'name': 'port', 'type': <class 'int'>, 'short': 'p',

dns_sd = None

doc_purpose = 'start the test webserver'

doc_usage = '[options]'

logger = None

name = 'serve'

shutdown(*signum=None*, *_frame=None*)
Shut down the server that is running detached.

class nikola.plugins.command.serve.**IPv6Server**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
Bases: **:class:'http.server.HTTPServer'**

An IPv6 HTTPServer.

address_family = 10

class nikola.plugins.command.serve.**OurHTTPRequestHandler**(*request*, *client_address*, *server*)
Bases: **:class:'http.server.SimpleHTTPRequestHandler'**

A request handler, modified for Nikola.

extensions_map = {'': 'text/plain', '.rxn': 'chemical/x-mdl-rxnfile', '.rss': 'application/x-rss+xml', '.txt': 'text/plain', '

log_message(**args*)
Log messages. Or not, depending on a setting.

quiet = False

send_head()
Send response code and MIME header.

This is common code for GET and HEAD commands.

Return value is either a file object (which has to be copied to the outputfile by the caller unless the command was HEAD, and must be closed by the caller under all circumstances), or None, in which case the caller has nothing further to do.

nikola.plugins.command.status module

Display site status.

class nikola.plugins.command.status.**CommandStatus**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Display site status.

cmd_options = [{'help': 'List all drafts', 'long': 'list-drafts', 'name': 'list_drafts', 'type': <class 'bool'>, 'short': 'd', 'd

doc_description = 'Show information about the posts and site deployment.'

doc_purpose = 'display site status'

doc_usage = '[-d|–list-drafts] [-m|–list-modified] [-p|–list-private] [-P|–list-published] [-s|–list-scheduled]'

human_time(*dt*)
Translate time into a human-friendly representation.

logger = None

name = 'status'

nikola.plugins.command.theme module

Manage themes.

class nikola.plugins.command.theme.**CommandTheme**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Manage themes.

cmd_options = [{'help': 'Install a theme.', 'long': 'install', 'name': 'install', 'type': <class 'str'>, 'short': 'i', 'default':

copy_template(*template*)
Copy the named template file from the parent to a local theme or to templates/.

do_install(*name*, *data*)
Download and install a theme.

do_install_deps(*url*, *name*)
Install themes and their dependencies.

do_uninstall(*name*)
Uninstall a theme.

doc_purpose = 'manage themes'

doc_usage = '[-u url] [-i theme_name] [-r theme_name] [-l] [–list-installed] [-g] [-n theme_name] [-c template_name]'

get_json(*url*)
Download the JSON file with all plugins.

get_path(*name*)
Get path for an installed theme.

json = None

list_available(*url*)
List all available themes.

list_installed()
List all installed themes.

name = 'theme'

new_theme(*name*, *engine*, *parent*, *create_legacy_meta=False*)
Create a new theme.

output_dir = 'themes'

nikola.plugins.command.version module

Print Nikola version.

class nikola.plugins.command.version.**CommandVersion**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.Command'**

Print Nikola version.

cmd_options = [{'help': 'Check for new versions.', 'long': 'check', 'name': 'check', 'type': <class 'bool'>, 'short': '', 'd

doc_purpose = 'print the Nikola version number'

doc_usage = '[–check]'

name = 'version'

needs_config = False

Module contents

Commands for Nikola.

nikola.plugins.compile package

Subpackages

nikola.plugins.compile.markdown package

Submodules

nikola.plugins.compile.markdown.mdx_gist module

Extension to Python Markdown for Embedded Gists (gist.github.com).

Basic Example:

Text of the gist: [:gist: 4747847]

Example with filename:

Text of the gist: [:gist: 4747847 zen.py]

Basic Example with hexidecimal id:

Text of the gist: [:gist: c4a43d6fdce612284ac0]

Example with hexidecimal id filename:

Text of the gist: [:gist: c4a43d6fdce612284ac0 cow.txt]

Example using reStructuredText syntax:

Text of the gist: .. gist:: 4747847 zen.py

Example using hexidecimal ID with reStructuredText syntax:

Text of the gist: .. gist:: c4a43d6fdce612284ac0

Example using hexidecimal ID and filename with reStructuredText syntax:

Text of the gist: .. gist:: c4a43d6fdce612284ac0 cow.txt

Error Case: non-existent Gist ID:

Text of the gist: [:gist: 0]

Error Case: non-existent file:

Text of the gist: [:gist: 4747847 doesntexist.py]

class nikola.plugins.compile.markdown.mdx_gist.**GistExtension**(*configs={}*)
Bases: **:class:'nikola.plugin_categories.MarkdownExtension'**, **:class:'markdown.extensions.Extension'**

Gist extension for Markdown.

extendMarkdown(*md*, *md_globals*)
Extend Markdown.

exception nikola.plugins.compile.markdown.mdx_gist.**GistFetchException**(*url*, *status_code*)
Bases: **:class:'Exception'**

Raised when attempt to fetch content of a Gist from github.com fails.

class nikola.plugins.compile.markdown.mdx_gist.**GistPattern**(*pattern*, *configs*)
Bases: **:class:'markdown.inlinepatterns.Pattern'**

InlinePattern for footnote markers in a document's body text.

get_raw_gist(*gist_id*)
Get raw gist text.

get_raw_gist_with_filename(*gist_id*, *filename*)
Get raw gist text for a filename.

handleMatch(*m*)
Handle pattern match.

nikola.plugins.compile.markdown.mdx_gist.**makeExtension**(*configs=None*)
Make Markdown extension.

nikola.plugins.compile.markdown.mdx_nikola module

Markdown Extension for Nikola.

- Specific post-processing.
- Strikethrough inline patterns.

class nikola.plugins.compile.markdown.mdx_nikola.**NikolaExtension**
Bases: **:class:'nikola.plugin_categories.MarkdownExtension'**, **:class:'markdown.extensions.Extension'**

Nikola Markdown extensions.

extendMarkdown(*md*, *md_globals*)
Extend markdown to Nikola flavours.

class nikola.plugins.compile.markdown.mdx_nikola.**NikolaPostProcessor**(*markdown_instance=None*)
Bases: **:class:'markdown.postprocessors.Postprocessor'**

Nikola-specific post-processing for Markdown.

run(*text*)
Run the postprocessor.

nikola.plugins.compile.markdown.mdx_nikola.**makeExtension**(*configs=None*)
Make extension.

nikola.plugins.compile.markdown.mdx_podcast module

Extension to Python Markdown for Embedded Audio.

Basic Example:

```
>>> import markdown
>>> text = "[podcast]https://archive.org/download/Rebeldes_Stereotipos/rs20120609_1.
↪mp3[/podcast]"
>>> html = markdown.markdown(text, [PodcastExtension()])
>>> print(html)
<p><audio controls=""><source src="https://archive.org/download/Rebeldes_Stereotipos/
↪rs20120609_1.mp3" type="audio/mpeg"></source></audio></p>
```

class nikola.plugins.compile.markdown.mdx_podcast.**PodcastExtension**(*configs={}*)
Bases: **:class:'nikola.plugin_categories.MarkdownExtension'**, **:class:'markdown.extensions.Extension'**

Podcast extension for Markdown.

extendMarkdown(*md*, *md_globals*)
Extend Markdown.

class nikola.plugins.compile.markdown.mdx_podcast.**PodcastPattern**(*pattern*, *configs*)
Bases: **:class:'markdown.inlinepatterns.Pattern'**

InlinePattern for footnote markers in a document's body text.

handleMatch(*m*)
Handle pattern matches.

nikola.plugins.compile.markdown.mdx_podcast.**makeExtension**(*configs=None*)
Make Markdown extension.

Module contents

Page compiler plugin for Markdown.

class nikola.plugins.compile.markdown.**CompileMarkdown**
Bases: **:class:'nikola.plugin_categories.PageCompiler'**

Compile Markdown into HTML.

compile(*source*, *dest*, *is_two_file=True*, *post=None*, *lang=None*)
Compile the source file into HTML and save as dest.

compile_string(*data*, *source_path=None*, *is_two_file=True*, *post=None*, *lang=None*)
Compile Markdown into HTML strings.

create_post(*path*, ***kw*)
Create a new post.

demote_headers = True

friendly_name = 'Markdown'

name = 'markdown'

read_metadata(*post*, *lang=None*)
Read the metadata from a post, and return a metadata dict.

set_site(*site*)
Set Nikola site.

site = None

supports_metadata = False

class nikola.plugins.compile.markdown.**ThreadLocalMarkdown**(*extensions*)
Bases: **:class:'_thread._local'**

Convert Markdown to HTML using per-thread Markdown objects.

See discussion in #2661.

convert(*data*)
Convert data to HTML and reset internal state.

nikola.plugins.compile.rest package

Submodules

nikola.plugins.compile.rest.chart module

Chart directive for reSTructuredText.

class nikola.plugins.compile.rest.chart.**Chart**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: **:class:'docutils.parsers.rst.Directive'**

reStructuredText extension for inserting charts as SVG.

Usage:

has_content = True

option_spec = {'classes': <function unchanged>, 'truncate_legend': <function unchanged>, 'x_labels': <function unch

required_arguments = 1

run()
Run the directive.

class nikola.plugins.compile.rest.chart.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for chart role.

name = 'rest_chart'

set_site(*site*)
Set Nikola site.

nikola.plugins.compile.rest.doc module

reST role for linking to other documents.

class nikola.plugins.compile.rest.doc.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for doc role.

name = 'rest_doc'

set_site(*site*)
Set Nikola site.

nikola.plugins.compile.rest.doc.**doc_role**(*name*, *rawtext*, *text*, *lineno*, *inliner*, *options={}*, *content=[]*)
Handle the doc role.

nikola.plugins.compile.rest.doc.**doc_shortcode**(**args*, ***kwargs*)
Implement the doc shortcode.

nikola.plugins.compile.rest.doc.**make_link_node**(*rawtext*, *text*, *url*, *options*)
Make a reST link node.

nikola.plugins.compile.rest.gist module

Gist directive for reStructuredText.

class nikola.plugins.compile.rest.gist.**GitHubGist**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: **:class:'docutils.parsers.rst.Directive'**

Embed GitHub Gist.

Usage:

or

final_argument_whitespace = True

get_raw_gist(*gistID*)
Get raw gist text.

get_raw_gist_with_filename(*gistID*, *filename*)
Get raw gist text for a filename.

has_content = False

option_spec = {'file': <function unchanged>}

optional_arguments = 1

required_arguments = 1

run()
Run the gist directive.

class nikola.plugins.compile.rest.gist.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for gist directive.

name = 'rest_gist'

set_site(*site*)
Set Nikola site.

nikola.plugins.compile.rest.listing module

Define and register a listing directive using the existing CodeBlock.

class nikola.plugins.compile.rest.listing.**CodeBlock**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: **:class:'docutils.parsers.rst.Directive'**

Parse and mark up content of a code block.

has_content = True

option_spec = {'name': <function unchanged>, 'number-lines': <function unchanged>, 'linenos': <function unchange

optional_arguments = 1

run()
Run code block directive.

class nikola.plugins.compile.rest.listing.**Listing**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: **:class:'docutils.parsers.rst.directives.misc.Include'**

Create a highlighted block of code from a file in listings/.

Usage:

assert_has_content()
Override check from superclass with nothing.

Listing has no content, override check from superclass.

get_code_from_file(*data*)
Create CodeBlock nodes from file object content.

has_content = False

option_spec = {'code': <function unchanged>, 'linenos': <function unchanged>, 'literal': <function flag>, 'number-lin

optional_arguments = 1

required_arguments = 1

run()
Run listing directive.

class nikola.plugins.compile.rest.listing.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for listing directive.

name = 'rest_listing'

set_site(*site*)
Set Nikola site.

nikola.plugins.compile.rest.media module

nikola.plugins.compile.rest.post_list module

Post list directive for reStructuredText.

class nikola.plugins.compile.rest.post_list.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for reST post-list directive.

name = 'rest_post_list'

set_site(*site*)
Set Nikola site.

class nikola.plugins.compile.rest.post_list.**PostList**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: **:class:'docutils.parsers.rst.Directive'**

Provide a reStructuredText directive to create a list of posts.

Directive Arguments None.

Directive Options lang, start, stop, reverse, sort, date, tags, categories, sections, slugs, post_type, template, id

Directive Content None.

The posts appearing in the list can be filtered by options. *List slicing* is provided with the *start*, *stop* and *reverse* options.

The following not required options are recognized:

start [integer] The index of the first post to show. A negative value like `-3` will show the *last* three posts in the post-list. Defaults to None.

stop [integer] The index of the last post to show. A value negative value like `-1` will show every post, but not the *last* in the post-list. Defaults to None.

reverse [flag] Reverse the order of the post-list. Defaults is to not reverse the order of posts.

sort [string] Sort post list by one of each post's attributes, usually `title` or a custom `priority`. Defaults to None (chronological sorting).

date [string] Show posts that match date range specified by this option. Format:

- comma-separated clauses (AND)
- clause: attribute comparison_operator value (spaces optional) * attribute: year, month, day, hour, month, second, weekday, isoweekday; or empty for full datetime * comparison_operator: == != <= >= < > * value: integer, 'now' or dateutil-compatible date input

tags [string [, string...]] Filter posts to show only posts having at least one of the `tags`. Defaults to None.

require_all_tags [flag] Change tag filter behaviour to show only posts that have all specified `tags`. Defaults to False.

categories [string [, string...]] Filter posts to show only posts having one of the `categories`. Defaults to None.

sections [string [, string...]] Filter posts to show only posts having one of the `sections`. Defaults to None.

slugs [string [, string...]] Filter posts to show only posts having at least one of the `slugs`. Defaults to None.

post_type (or type) [string] Show only `posts`, `pages` or `all`. Replaces `all`. Defaults to `posts`.

lang [string] The language of post *titles* and *links*. Defaults to default language.

template [string] The name of an alternative template to render the post-list. Defaults to `post_list_directive.tmpl`

id [string] A manual id for the post list. Defaults to a random name composed by 'post_list_' + uuid.uuid4().hex.

option_spec = {'categories': <function unchanged>, 'slugs': <function unchanged>, 'sort': <function unchanged>, 'ty

run()
Run post-list directive.

nikola.plugins.compile.rest.slides module

nikola.plugins.compile.rest.soundcloud module

SoundCloud directive for reStructuredText.

class nikola.plugins.compile.rest.soundcloud.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for soundclound directive.

name = 'rest_soundcloud'

set_site(*site*)
Set Nikola site.

class nikola.plugins.compile.rest.soundcloud.**SoundCloud**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: **:class:'docutils.parsers.rst.Directive'**

reST extension for inserting SoundCloud embedded music.

Usage:

check_content()
Emit a deprecation warning if there is content.

has_content = True

option_spec = {'width': <function positive_int>, 'align': <function _align_choice>, 'height': <function positive_int>}

preslug = 'tracks'

required_arguments = 1

run()
Run the soundcloud directive.

class nikola.plugins.compile.rest.soundcloud.**SoundCloudPlaylist**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: **:class:'nikola.plugins.compile.rest.soundcloud.SoundCloud'**

reST directive for SoundCloud playlists.

preslug = 'playlists'

nikola.plugins.compile.rest.thumbnail module

Thumbnail directive for reStructuredText.

class nikola.plugins.compile.rest.thumbnail.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for thumbnail directive.

name = 'rest_thumbnail'

set_site(*site*)
Set Nikola site.

class nikola.plugins.compile.rest.thumbnail.**Thumbnail**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: **:class:'docutils.parsers.rst.directives.images.Figure'**

Thumbnail directive for reST.

align(*argument*)
Return thumbnail alignment.

figwidth_value(*argument*)
Return figure width.

has_content = True

option_spec = {'alt': <function unchanged>, 'figclass': <function class_option>, 'figwidth': <function Thumbnail.figwi

run()
Run the thumbnail directive.

nikola.plugins.compile.rest.vimeo module

Vimeo directive for reStructuredText.

class nikola.plugins.compile.rest.vimeo.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for vimeo reST directive.

name = 'rest_vimeo'

set_site(*site*)
Set Nikola site.

class nikola.plugins.compile.rest.vimeo.**Vimeo**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: **:class:'docutils.parsers.rst.Directive'**

reST extension for inserting vimeo embedded videos.

Usage:

check_content()
Check if content exists.

check_modules()
Check modules.

has_content = True

option_spec = {'width': <function positive_int>, 'align': <function _align_choice>, 'height': <function positive_int>}

request_size = True

required_arguments = 1

run()
Run the vimeo directive.

set_video_size()
Set video size.

nikola.plugins.compile.rest.youtube module

YouTube directive for reStructuredText.

class nikola.plugins.compile.rest.youtube.**Plugin**
Bases: **:class:'nikola.plugin_categories.RestExtension'**

Plugin for the youtube directive.

name = 'rest_youtube'

set_site(*site*)
Set Nikola site.

class nikola.plugins.compile.rest.youtube.**Youtube**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: **:class:'docutils.parsers.rst.Directive'**

reST extension for inserting youtube embedded videos.

Usage:

check_content()
Check if content exists.

has_content = True

option_spec = {'width': <function positive_int>, 'align': <function _align_choice>, 'height': <function positive_int>}

required_arguments = 1

run()
Run the youtube directive.

Module contents

reStructuredText compiler for Nikola.

class nikola.plugins.compile.rest.**CompileRest**
Bases: **:class:'nikola.plugin_categories.PageCompiler'**

Compile reStructuredText into HTML.

compile(*source*, *dest*, *is_two_file=True*, *post=None*, *lang=None*)
Compile the source file into HTML and save as dest.

compile_string(*data*, *source_path=None*, *is_two_file=True*, *post=None*, *lang=None*)
Compile reST into HTML strings.

create_post(*path*, ***kw*)
Create a new post.

demote_headers = True

friendly_name = 'reStructuredText'

logger = None

metadata_conditions = [(<MetaCondition.config_bool: 1>, 'USE_REST_DOCINFO_METADATA')]

name = 'rest'

read_metadata(*post*, *lang=None*)
Read the metadata from a post, and return a metadata dict.

set_site(*site*)
Set Nikola site.

supports_metadata = True

class nikola.plugins.compile.rest.**NikolaReader**(**args*, ***kwargs*)
Bases: **:class:'docutils.readers.standalone.Reader'**

Nikola-specific docutils reader.

get_transforms()
Get docutils transforms.

new_document()
Create and return a new empty document tree (root node).

class nikola.plugins.compile.rest.**RemoveDocinfo**(*document*, *startnode=None*)
Bases: **:class:'docutils.transforms.Transform'**

Remove docinfo nodes.

apply()
Remove docinfo nodes.

default_priority = 870

nikola.plugins.compile.rest.**add_node**(*node*, *visit_function=None*, *depart_function=None*)
Register a Docutils node class.

This function is completely optional. It is a same concept as Sphinx add_node function.

For example:

```
class Plugin(RestExtension):

    name = "rest_math"

    def set_site(self, site):
        self.site = site
        directives.register_directive('math', MathDirective)
        add_node(MathBlock, visit_Math, depart_Math)
        return super(Plugin, self).set_site(site)

class MathDirective(Directive):
    def run(self):
        node = MathBlock()
```

```
        return [node]

class Math(docutils.nodes.Element): pass

def visit_Math(self, node):
    self.body.append(self.starttag(node, 'math'))

def depart_Math(self, node):
    self.body.append('</math>')
```

For full example, you can refer to Microdata plugin

nikola.plugins.compile.rest.**get_observer**(*settings*)
Return an observer for the docutils Reporter.

nikola.plugins.compile.rest.**rst2html**(*source*, *source_path=None*, *source_class=<class 'docutils.io.StringInput'>*, *destination_path=None*, *reader=None*, *parser=None*, *parser_name='restructuredtext'*, *writer=None*, *writer_name='html'*, *settings=None*, *settings_spec=None*, *settings_overrides=None*, *config_section=None*, *enable_exit_status=None*, *logger=None*, *l_add_ln=0*, *transforms=None*, *no_title_transform=False*)
Set up & run a `Publisher`, and return a dictionary of document parts.

Dictionary keys are the names of parts, and values are Unicode strings; encoding is up to the client. For programmatic use with string I/O.

For encoded string input, be sure to set the 'input_encoding' setting to the desired encoding. Set it to 'unicode' for unencoded Unicode string input. Here's how:

```
publish_parts(..., settings_overrides={'input_encoding': 'unicode'})
```

For a description of the parameters, see *publish_programmatically*.

WARNING: *reader* should be None (or NikolaReader()) if you want Nikola to report reStructuredText syntax errors.

nikola.plugins.compile.rest.**shortcode_role**(*name*, *rawtext*, *text*, *lineno*, *inliner*, *options={}*, *content=[]*)
Return a shortcode role that passes through raw inline HTML.

Submodules

nikola.plugins.compile.html module

Page compiler plugin for HTML source files.

class nikola.plugins.compile.html.**CompileHtml**
Bases: **:class:'nikola.plugin_categories.PageCompiler'**

Compile HTML into HTML.

compile(*source*, *dest*, *is_two_file=True*, *post=None*, *lang=None*)
Compile the source file into HTML and save as dest.

compile_string(*data*, *source_path=None*, *is_two_file=True*, *post=None*, *lang=None*)
Compile HTML into HTML strings, with shortcode support.

create_post (*path*, ***kw*)
Create a new post.

friendly_name = 'HTML'

name = 'html'

read_metadata (*post*, *file_metadata_regexp=None*, *unslugify_titles=False*, *lang=None*)
Read the metadata from a post's meta tags, and return a metadata dict.

supports_metadata = True

nikola.plugins.compile.ipynb module

Page compiler plugin for nbconvert.

class nikola.plugins.compile.ipynb.**CompileIPynb**
Bases: **:class:'nikola.plugin_categories.PageCompiler'**

Compile IPynb into HTML.

compile (*source*, *dest*, *is_two_file=False*, *post=None*, *lang=None*)
Compile the source file into HTML and save as dest.

compile_string (*data*, *source_path=None*, *is_two_file=True*, *post=None*, *lang=None*)
Compile notebooks into HTML strings.

create_post (*path*, ***kw*)
Create a new post.

default_kernel = 'python3'

demote_headers = True

friendly_name = 'Jupyter Notebook'

name = 'ipynb'

read_metadata (*post*, *lang=None*)
Read metadata directly from ipynb file.

As ipynb files support arbitrary metadata as json, the metadata used by Nikola will be assume to be in the 'nikola' subfield.

set_site (*site*)
Set Nikola site.

supports_metadata = True

nikola.plugins.compile.ipynb.**get_default_jupyter_config**()
Search default jupyter configuration location paths.

Return dictionary from configuration json files.

nikola.plugins.compile.pandoc module

Page compiler plugin for pandoc.

You will need, of course, to install pandoc

class nikola.plugins.compile.pandoc.**CompilePandoc**
Bases: **:class:'nikola.plugin_categories.PageCompiler'**

Compile markups into HTML using pandoc.

compile(*source*, *dest*, *is_two_file=True*, *post=None*, *lang=None*)
Compile the source file into HTML and save as dest.

compile_string(*data*, *source_path=None*, *is_two_file=True*, *post=None*, *lang=None*)
Compile into HTML strings.

create_post(*path*, ***kw*)
Create a new post.

friendly_name = 'pandoc'

name = 'pandoc'

set_site(*site*)
Set Nikola site.

nikola.plugins.compile.php module

Page compiler plugin for PHP.

class nikola.plugins.compile.php.**CompilePhp**
Bases: **:class:'nikola.plugin_categories.PageCompiler'**

Compile PHP into PHP.

compile(*source*, *dest*, *is_two_file=True*, *post=None*, *lang=None*)
Compile the source file into HTML and save as dest.

compile_string(*data*, *source_path=None*, *is_two_file=True*, *post=None*, *lang=None*)
Compile PHP into HTML strings.

create_post(*path*, ***kw*)
Create a new post.

extension()
Return extension used for PHP files.

friendly_name = 'PHP'

name = 'php'

Module contents

Compilers for Nikola.

nikola.plugins.misc package

Submodules

nikola.plugins.misc.scan_posts module

The default post scanner.

class nikola.plugins.misc.scan_posts.**ScanPosts**
Bases: **:class:'nikola.plugin_categories.PostScanner'**

Scan posts in the site.

name = 'scan_posts'

scan()
Create list of posts from POSTS and PAGES options.

supported_extensions()
Return a list of supported file extensions, or None if such a list isn't known beforehand.

Module contents

Miscellaneous Nikola plugins.

nikola.plugins.task package

Subpackages

nikola.plugins.task.sitemap package

Module contents

Generate a sitemap.

class nikola.plugins.task.sitemap.**Sitemap**
Bases: **:class:'nikola.plugin_categories.LateTask'**

Generate a sitemap.

gen_tasks()
Generate a sitemap.

get_lastmod(*p*)
Get last modification date.

name = 'sitemap'

nikola.plugins.task.sitemap.**get_base_path**(*base*)
Return the path of a base URL if it contains one.

```
>>> get_base_path('http://some.site') == '/'
True
>>> get_base_path('http://some.site/') == '/'
True
>>> get_base_path('http://some.site/some/sub-path') == '/some/sub-path/'
True
>>> get_base_path('http://some.site/some/sub-path/') == '/some/sub-path/'
True
```

Submodules

nikola.plugins.task.archive module

Classify the posts in archives.

class nikola.plugins.task.archive.**Archive**
Bases: **:class:'nikola.plugin_categories.Taxonomy'**

Classify the post archives.

add_other_languages_variable = True

always_disable_rss = True

apply_to_pages = False

apply_to_posts = True

classification_name = 'archive'

classify (*post*, *lang*)
Classify the given post for the given language.

extract_hierarchy (*classification*)
Given a classification, return a list of parts in the hierarchy.

generate_atom_feeds_for_post_lists = False

get_classification_friendly_name (*classification*, *lang*, *only_last_component=False*)
Extract a friendly name from the classification.

get_implicit_classifications (*lang*)
Return a list of classification strings which should always appear in posts_per_classification.

get_other_language_variants (*classification*, *lang*, *classifications_per_language*)
Return a list of variants of the same classification in other languages.

get_path (*classification*, *lang*, *dest_type='page'*)
Return a path for the given classification.

has_hierarchy = True

include_posts_from_subhierarchies = True

include_posts_into_hierarchy_root = True

minimum_post_count_per_classification_in_overview = 1

more_than_one_classifications_per_post = False

name = 'classify_archive'

omit_empty_classifications = False

overview_page_variable_name = 'archive'

path_handler_docstrings = {'archive': 'Link to archive path, name is the year.\n\n Example:\n\n link://archive/201

postprocess_posts_per_classification (*posts_per_classification_per_language*, *flat_hierarchy_per_lang=None*, *hierarchy_lookup_per_lang=None*)
Rearrange, modify or otherwise use the list of posts per classification and per language.

provide_context_and_uptodate (*classification*, *lang*, *node=None*)
Provide data for the context and the uptodate list for the list of the given classifiation.

recombine_classification_from_hierarchy (*hierarchy*)
Given a list of parts in the hierarchy, return the classification string.

set_site (*site*)
Set Nikola site.

should_generate_classification_page (*classification*, *post_list*, *lang*)
Only generates list of posts for classification if this function returns True.

sort_classifications (*classifications*, *lang*, *level=None*)
Sort the given list of classification strings.

subcategories_list_template = 'list.tmpl'

template_for_classification_overview = None

nikola.plugins.task.authors module

Render the author pages and feeds.

class nikola.plugins.task.authors.**ClassifyAuthors**
Bases: **:class:'nikola.plugin_categories.Taxonomy'**

Classify the posts by authors.

add_other_languages_variable = True

apply_to_pages = False

apply_to_posts = True

classification_name = 'author'

classify (*post*, *lang*)
Classify the given post for the given language.

generate_atom_feeds_for_post_lists = False

get_classification_friendly_name (*classification*, *lang*, *only_last_component=False*)
Extract a friendly name from the classification.

get_other_language_variants (*classification*, *lang*, *classifications_per_language*)
Return a list of variants of the same author in other languages.

get_overview_path (*lang*, *dest_type='page'*)
Return a path for the list of all classifications.

get_path (*classification*, *lang*, *dest_type='page'*)
Return a path for the given classification.

has_hierarchy = False

is_enabled (*lang=None*)
Return True if this taxonomy is enabled, or False otherwise.

minimum_post_count_per_classification_in_overview = 1

more_than_one_classifications_per_post = False

name = 'classify_authors'

omit_empty_classifications = False

overview_page_variable_name = 'authors'

path_handler_docstrings = {'author_index': ' Link to the authors index.\n\n Example:\n\n link://authors/ => /auth

postprocess_posts_per_classification(*posts_per_classification_per_language*, *flat_hierarchy_per_lang=None*, *hierarchy_lookup_per_lang=None*)
Rearrange, modify or otherwise use the list of posts per classification and per language.

provide_context_and_uptodate(*classification*, *lang*, *node=None*)
Provide data for the context and the uptodate list for the list of the given classifiation.

provide_overview_context_and_uptodate(*lang*)
Provide data for the context and the uptodate list for the list of all classifiations.

set_site(*site*)
Set Nikola site.

template_for_classification_overview = 'authors.tmpl'

nikola.plugins.task.bundles module

Bundle assets using WebAssets.

class nikola.plugins.task.bundles.**BuildBundles**
Bases: **:class:'nikola.plugin_categories.LateTask'**

Bundle assets using WebAssets.

gen_tasks()
Bundle assets using WebAssets.

name = 'create_bundles'

set_site(*site*)
Set Nikola site.

nikola.plugins.task.bundles.**get_theme_bundles**(*themes*)
Given a theme chain, return the bundle definitions.

nikola.plugins.task.copy_assets module

Copy theme assets into output.

class nikola.plugins.task.copy_assets.**CopyAssets**
Bases: **:class:'nikola.plugin_categories.Task'**

Copy theme assets into output.

gen_tasks()
Create tasks to copy the assets of the whole theme chain.

If a file is present on two themes, use the version from the "youngest" theme.

name = 'copy_assets'

nikola.plugins.task.copy_files module

Copy static files into the output folder.

class nikola.plugins.task.copy_files.**CopyFiles**
Bases: **:class:'nikola.plugin_categories.Task'**

Copy static files into the output folder.

gen_tasks()
Copy static files into the output folder.

name = 'copy_files'

nikola.plugins.task.galleries module

Render image galleries.

class nikola.plugins.task.galleries.**Galleries**
Bases: **:class:'nikola.plugin_categories.Task'**, **:class:'nikola.image_processing.ImageProcessor'**

Render image galleries.

create_galleries()
Given a list of galleries, create the output folders.

create_galleries_paths()
Given a list of galleries, put their paths into self.gallery_links.

create_target_images(*img*, *input_path*)
Copy images to output.

dates = {}

find_galleries()
Find all galleries to be processed according to conf.py.

gallery_global_path(*name*, *lang*)
Link to the global gallery path, which contains all the images in galleries.

There is only one copy of an image on multilingual blogs, in the site root.

link://gallery_global/london => /galleries/trips/london/index.html

link://gallery_global/trips/london => /galleries/trips/london/index.html

(a gallery link could lead to eg. /en/galleries/trips/london/index.html)

gallery_path(*name*, *lang*)
Link to an image gallery's path.

It will try to find a gallery with that name if it's not ambiguous or with that path. For example:

link://gallery/london => /galleries/trips/london/index.html

link://gallery/trips/london => /galleries/trips/london/index.html

gallery_rss(*img_list*, *dest_img_list*, *img_titles*, *lang*, *permalink*, *output_path*, *title*)
Create a RSS showing the latest images in the gallery.

This doesn't use generic_rss_renderer because it doesn't involve Post objects.

gallery_rss_path(*name*, *lang*)
Link to an image gallery's RSS feed.

It will try to find a gallery with that name if it's not ambiguous or with that path. For example:

link://gallery_rss/london => /galleries/trips/london/rss.xml

link://gallery_rss/trips/london => /galleries/trips/london/rss.xml

gen_tasks()
Render image galleries.

get_excluded_images(*gallery_path*)
Get list of excluded images.

get_image_list(*gallery_path*)
Get list of included images.

name = 'render_galleries'

parse_index(*gallery*, *input_folder*, *output_folder*)
Return a Post object if there is an index.txt.

remove_excluded_image(*img*, *input_folder*)
Remove excluded images.

render_gallery_index(*template_name*, *output_name*, *context*, *img_list*, *img_titles*, *thumbs*, *file_dep*)
Build the gallery index.

set_site(*site*)
Set Nikola site.

nikola.plugins.task.gzip module

Create gzipped copies of files.

class nikola.plugins.task.gzip.**GzipFiles**
Bases: **:class:'nikola.plugin_categories.TaskMultiplier'**

If appropiate, create tasks to create gzipped versions of files.

is_default = True

name = 'gzip'

process(*task*, *prefix*)
Process tasks.

nikola.plugins.task.gzip.**create_gzipped_copy**(*in_path*, *out_path*, *command=None*)
Create gzipped copy of in_path and save it as out_path.

nikola.plugins.task.indexes module

Render the blog's main index.

class nikola.plugins.task.indexes.**Indexes**
Bases: **:class:'nikola.plugin_categories.Taxonomy'**

Classify for the blog's main index.

apply_to_pages = False

apply_to_posts = True

classification_name = 'index'

classify(*post*, *lang*)
Classify the given post for the given language.

get_classification_friendly_name(*classification*, *lang*, *only_last_component=False*)
Extract a friendly name from the classification.

get_implicit_classifications(*lang*)
Return a list of classification strings which should always appear in posts_per_classification.

get_path(*classification*, *lang*, *dest_type='page'*)
Return a path for the given classification.

has_hierarchy = False

more_than_one_classifications_per_post = False

name = 'classify_indexes'

omit_empty_classifications = False

overview_page_variable_name = None

path_handler_docstrings = {'index_rss': 'A link to the RSS feed path.\n\nExample:\n\nlink://rss => /blog/rss.xml',

provide_context_and_uptodate(*classification*, *lang*, *node=None*)
Provide data for the context and the uptodate list for the list of the given classifiation.

set_site(*site*)
Set Nikola site.

should_generate_classification_page(*classification*, *post_list*, *lang*)
Only generates list of posts for classification if this function returns True.

should_generate_rss_for_classification_page(*classification*, *post_list*, *lang*)
Only generates RSS feed for list of posts for classification if this function returns True.

show_list_as_index = True

template_for_classification_overview = None

template_for_single_list = 'index.tmpl'

nikola.plugins.task.listings module

Render code listings.

class nikola.plugins.task.listings.**Listings**
Bases: **:class:'nikola.plugin_categories.Task'**

Render code listings.

gen_tasks()
Render pretty code listings.

listing_path(*namep*, *lang*)
Return a link to a listing.

It will try to use the file name if it's not ambiguous, or the file path.

Example:

link://listing/hello.py => /listings/tutorial/hello.py.html

link://listing/tutorial/hello.py => /listings/tutorial/hello.py.html

listing_source_path(*name*, *lang*)
Return a link to the source code for a listing.

It will try to use the file name if it's not ambiguous, or the file path.

Example:

link://listing_source/hello.py => /listings/tutorial/hello.py

link://listing_source/tutorial/hello.py => /listings/tutorial/hello.py

name = 'render_listings'

register_output_name(*input_folder*, *rel_name*, *rel_output_name*)
Register proper and improper file mappings.

set_site(*site*)
Set Nikola site.

nikola.plugins.task.pages module

Render pages into output.

class nikola.plugins.task.pages.**RenderPages**
Bases: **:class:'nikola.plugin_categories.Task'**

Render pages into output.

gen_tasks()
Build final pages from metadata and HTML fragments.

name = 'render_pages'

nikola.plugins.task.posts module

Build HTML fragments from metadata and text.

class nikola.plugins.task.posts.**RenderPosts**
Bases: **:class:'nikola.plugin_categories.Task'**

Build HTML fragments from metadata and text.

dependence_on_timeline(*post*, *lang*)
Check if a post depends on the timeline.

gen_tasks()
Build HTML fragments from metadata and text.

name = 'render_posts'

nikola.plugins.task.posts.**update_deps**(*post*, *lang*, *task*)
Update file dependencies as they might have been updated during compilation.

This is done for example by the ReST page compiler, which writes its dependencies into a .dep file. This file is read and incorporated when calling post.fragment_deps(), and only available /after/ compiling the fragment.

nikola.plugins.task.py3_switch module

nikola.plugins.task.redirect module

Generate redirections.

class nikola.plugins.task.redirect.**Redirect**
Bases: **:class:'nikola.plugin_categories.Task'**

Generate redirections.

gen_tasks()
Generate redirections tasks.

name = 'redirect'

nikola.plugins.task.robots module

Generate a robots.txt file.

class nikola.plugins.task.robots.**RobotsFile**
Bases: **:class:'nikola.plugin_categories.LateTask'**

Generate a robots.txt file.

gen_tasks()
Generate a robots.txt file.

name = 'robots_file'

nikola.plugins.task.rss module

nikola.plugins.task.scale_images module

Resize images and create thumbnails for them.

class nikola.plugins.task.scale_images.**ScaleImage**
Bases: **:class:'nikola.plugin_categories.Task'**, **:class:'nikola.image_processing.ImageProcessor'**

Resize images and create thumbnails for them.

gen_tasks()
Copy static files into the output folder.

name = 'scale_images'

process_image(*src*, *dst*, *thumb*)
Resize an image.

process_tree(*src*, *dst*)
Process all images in a src tree and put the (possibly) rescaled images in the dst folder.

set_site(*site*)
Set Nikola site.

nikola.plugins.task.sources module

Copy page sources into the output.

class nikola.plugins.task.sources.**Sources**
Bases: **:class:'nikola.plugin_categories.Task'**

Copy page sources into the output.

gen_tasks()
Publish the page sources into the output.

name = 'render_sources'

nikola.plugins.task.tags module

Render the tag pages and feeds.

class nikola.plugins.task.tags.**ClassifyTags**
Bases: **:class:'nikola.plugin_categories.Taxonomy'**

Classify the posts by tags.

add_other_languages_variable = True

always_disable_rss = False

apply_to_pages = False

apply_to_posts = True

classification_name = 'tag'

classify (*post*, *lang*)
Classify the given post for the given language.

generate_atom_feeds_for_post_lists = True

get_classification_friendly_name (*classification*, *lang*, *only_last_component=False*)
Extract a friendly name from the classification.

get_other_language_variants (*classification*, *lang*, *classifications_per_language*)
Return a list of variants of the same tag in other languages.

get_overview_path (*lang*, *dest_type='page'*)
Return a path for the list of all classifications.

get_path (*classification*, *lang*, *dest_type='page'*)
Return a path for the given classification.

has_hierarchy = False

is_enabled (*lang=None*)
Return True if this taxonomy is enabled, or False otherwise.

more_than_one_classifications_per_post = True

name = 'classify_tags'

omit_empty_classifications = True

overview_page_items_variable_name = 'items'

overview_page_variable_name = 'tags'

path_handler_docstrings = {'tag_rss': "A link to a tag's RSS feed.\n\nExample:\n\nlink://tag_rss/cats => /tags/cats

postprocess_posts_per_classification (*posts_per_classification_per_language*, *flat_hierarchy_per_lang=None*, *hierarchy_lookup_per_lang=None*)
Rearrange, modify or otherwise use the list of posts per classification and per language.

provide_context_and_uptodate (*classification*, *lang*, *node=None*)
Provide data for the context and the uptodate list for the list of the given classifiation.

provide_overview_context_and_uptodate (*lang*)
Provide data for the context and the uptodate list for the list of all classifiations.

set_site (*site*)
Set site, which is a Nikola instance.

show_list_as_subcategories_list = False

slugify_tag_name(*name*, *lang*)
Slugify a tag name.

template_for_classification_overview = 'tags.tmpl'

Module contents

Tasks for Nikola.

nikola.plugins.template package

Submodules

nikola.plugins.template.jinja module

Jinja template handler.

class nikola.plugins.template.jinja.**JinjaTemplates**
Bases: **:class:'nikola.plugin_categories.TemplateSystem'**

Support for Jinja2 templates.

create_lookup()
Create a template lookup.

dependency_cache = {}

get_deps(*filename*)
Return paths to dependencies for the template loaded from filename.

get_string_deps(*text*)
Find dependencies for a template string.

get_template_path(*template_name*)
Get the path to a template or return None.

inject_directory(*directory*)
Add a directory to the lookup and recreate it if it's not there yet.

lookup = None

name = 'jinja'

per_file_cache = {}

render_template(*template_name*, *output_name*, *context*)
Render the template into output_name using context.

render_template_to_string(*template*, *context*)
Render template to a string using context.

set_directories(*directories*, *cache_folder*)
Create a new template lookup with set directories.

set_site(*site*)
Set the Nikola site.

template_deps(*template_name*)
Generate list of dependencies for a template.

nikola.plugins.template.mako module

Mako template handler.

class nikola.plugins.template.mako.**MakoTemplates**
Bases: **:class:‘nikola.plugin_categories.TemplateSystem‘**

Support for Mako templates.

cache = {}

cache_dir = None

create_lookup()
Create a template lookup.

directories = []

filters = {}

get_deps(*filename*)
Get paths to dependencies for a template.

get_string_deps(*text*, *filename=None*)
Find dependencies for a template string.

get_template_path(*template_name*)
Get the path to a template or return None.

inject_directory(*directory*)
Add a directory to the lookup and recreate it if it’s not there yet.

lookup = None

name = ‘mako’

render_template(*template_name*, *output_name*, *context*)
Render the template into output_name using context.

render_template_to_string(*template*, *context*)
Render template to a string using context.

set_directories(*directories*, *cache_folder*)
Create a new template lookup with set directories.

set_site(*site*)
Set the Nikola site.

template_deps(*template_name*)
Generate list of dependencies for a template.

nikola.plugins.template.mako.**striphtml**(*text*)
Strip HTML tags from text.

Module contents

Default template engines for Nikola.

Submodules

nikola.plugins.basic_import module

Mixin for importer plugins.

class nikola.plugins.basic_import.**ImportMixin**
Bases: **:class:‘object‘**

Mixin with common used methods.

cmd_options = [{‘long’: ‘output-folder’, ‘name’: ‘output_folder’, ‘short’: ‘o’, ‘default’: ‘new_site’, ‘help’: ‘Location to

static configure_redirections(*url_map*, *base_dir=’‘*)
Configure redirections from an url_map.

doc_purpose = ‘import a dump from a different engine.’

doc_usage = ‘[options] export_file’

generate_base_site()
Generate a base Nikola site.

classmethod get_channel_from_file(*filename*)
Get channel from XML file.

get_configuration_output_path()
Get path for the output configuration file.

name = ‘import_mixin’

needs_config = False

static populate_context(*channel*)
Populate context with settings.

classmethod transform_content(*content*)
Transform content to a Nikola-friendly format.

static write_configuration(*filename*, *rendered_template*)
Write the configuration file.

classmethod write_content(*filename*, *content*, *rewrite_html=True*)
Write content to file.

write_metadata(*filename*, *title*, *slug*, *post_date*, *description*, *tags*, ***kwargs*)
Write metadata to meta file.

classmethod write_post(*filename*, *content*, *headers*, *compiler*, *rewrite_html=True*)
Ask the specified compiler to write the post to disk.

static write_urlmap_csv(*output_file*, *url_map*)
Write urlmap to csv file.

nikola.plugins.basic_import.**replacer**(*dst*)
Replace links.

Module contents

Plugins for Nikola.

Submodules

nikola.filters module

Utility functions to help run filters on files.

All filters defined in this module are registered in Nikola.__init__.

nikola.filters.**add_header_permalinks**(*fname*, *xpath_list=None*, *file_blacklist=None*)
Post-process HTML via lxml to add header permalinks Sphinx-style.

nikola.filters.**apply_to_binary_file**(*f*)
Apply a filter to a binary file.

Take a function f that transforms a data argument, and returns a function that takes a filename and applies f to the contents, in place. Reads files in binary mode.

nikola.filters.**apply_to_text_file**(*f*)
Apply a filter to a text file.

Take a function f that transforms a data argument, and returns a function that takes a filename and applies f to the contents, in place. Reads files in UTF-8.

nikola.filters.**closure_compiler**(*infile*, *executable='closure-compiler'*)
Run closure-compiler on a file.

nikola.filters.**cssminify**(*data*)
Minify CSS using http://cssminifier.com/.

nikola.filters.**deduplicate_ids**(*data*, *top_classes=None*)
Post-process HTML via lxml to deduplicate IDs.

nikola.filters.**html5lib_minify**(*data*)
Minify with html5lib.

nikola.filters.**html5lib_xmllike**(*data*)
Transform document to an XML-like form with html5lib.

nikola.filters.**html_tidy_mini**(*infile*, *executable='tidy5'*)
Run HTML tidy with minimal settings.

nikola.filters.**html_tidy_nowrap**(*infile*, *executable='tidy5'*)
Run HTML Tidy without line wrapping.

nikola.filters.**html_tidy_withconfig**(*infile*, *executable='tidy5'*)
Run HTML Tidy with tidy5.conf as config file.

nikola.filters.**html_tidy_wrap**(*infile*, *executable='tidy5'*)
Run HTML Tidy with line wrapping.

nikola.filters.**html_tidy_wrap_attr**(*infile*, *executable='tidy5'*)
Run HTML tidy with line wrapping and attribute indentation.

nikola.filters.**jpegoptim**(*infile*, *executable='jpegoptim'*)
Run jpegoptim on a file.

nikola.filters.**jpegoptim_progressive**(*infile*, *executable='jpegoptim'*)
Run jpegoptim on a file and convert to progressive.

nikola.filters.**jsminify**(*data*)
Minify JS using http://javascript-minifier.com/.

nikola.filters.**jsonminify**(*data*)
Minify JSON files (strip whitespace and use minimal separators).

nikola.filters.**list_replace**(*the_list*, *find*, *replacement*)
Replace all occurrences of `find` with `replacement` in `the_list`.

nikola.filters.**minify_lines**(*data*)
Do nothing – deprecated filter.

nikola.filters.**normalize_html**(*data*)
Pass HTML through LXML to clean it up, if possible.

nikola.filters.**optipng**(*infile*, *executable='optipng'*)
Run optipng on a file.

nikola.filters.**php_template_injection**(*data*)
Insert PHP code into Nikola templates.

nikola.filters.**runinplace**(*command*, *infile*)
Run a command in-place on a file.

command is a string of the form: "commandname %1 %2" and it will be execed with infile as %1 and a temporary file as %2. Then, that temporary file will be moved over %1.

Example usage:

runinplace("yui-compressor %1 -o %2", "myfile.css")

That will replace myfile.css with a minified version.

You can also supply command as a list.

nikola.filters.**typogrify**(*data*)
Prettify text with typogrify.

nikola.filters.**typogrify_oldschool**(*data*)
Prettify text with typogrify.

nikola.filters.**typogrify_sans_widont**(*data*)
Prettify text with typogrify, skipping the widont filter.

nikola.filters.**xmlminify**(*data*)
Minify XML files (strip whitespace and use minimal separators).

nikola.filters.**yui_compressor**(*infile*, *executable=None*)
Run YUI Compressor on a file.

nikola.image_processing module

Process images.

class nikola.image_processing.**ImageProcessor**
Bases: **:class:'object'**

Apply image operations.

filter_exif(*exif*, *whitelist*)
Filter EXIF data as described in the documentation.

image_date(*src*)
Try to figure out the date of the image.

image_ext_list_builtin = ['.jpg', '.png', '.jpeg', '.gif', '.svg', '.svgz', '.bmp', '.tiff']

resize_image(*src*, *dst*, *max_size*, *bigger_panoramas=True*, *preserve_exif_data=False*, *exif_whitelist={}*)
Make a copy of the image in the requested size.

resize_svg(*src*, *dst*, *max_size*, *bigger_panoramas*)
Make a copy of an svg at the requested size.

nikola.nikola module

The main Nikola site object.

class nikola.nikola.**Nikola**(**config*)
Bases: **:class:'object'**

Class that handles site generation.

Takes a site config as argument on creation.

GLOBAL_CONTEXT
Initialize some parts of GLOBAL_CONTEXT only when it's queried.

MESSAGES

THEMES

abs_link(*dst*, *protocol_relative=False*)
Get an absolute link.

apply_shortcodes(*data*, *filename=None*, *lang=None*, *extra_context=None*)
Apply shortcodes from the registry on data.

apply_shortcodes_uuid(*data*, *_shortcodes*, *filename=None*, *lang=None*, *extra_context=None*)
Apply shortcodes from the registry on data.

atom_feed_renderer(*lang*, *posts*, *output_path*, *filters*, *extra_context*)
Render Atom feeds and archives with lists of posts.

Feeds are considered archives when no future updates to them are expected.

category_path_to_category_name(*category_path*)
Translate a category path to a category name.

clean_task_paths(*task*)
Normalize target paths in the task.

file_exists(*path*, *not_empty=False*)
Check if the file exists. If not_empty is True, it also must not be empty.

filename_path(*name*, *lang*)
Link to post or page by source filename.

Example:

link://filename/manual.txt => /docs/handbook.html

gen_tasks(*name*, *plugin_category*, *doc=''*)
Generate tasks.

generic_index_renderer(*lang*, *posts*, *indexes_title*, *template_name*, *context_source*, *kw*, *basename*, *page_link*, *page_path*, *additional_dependencies=[]*)
Create an index page.

lang: The language posts: A list of posts indexes_title: Title template_name: Name of template file context_source: This will be copied and extended and used as every

page's context

kw: An extended version will be used for uptodate dependencies basename: Basename for task page_link: A function accepting an index i, the displayed page number,

the number of pages, and a boolean force_addition which creates a link to the i-th page (where i ranges between 0 and num_pages-1). The displayed page (between 1 and num_pages) is the number (optionally) displayed as 'page %d' on the rendered page. If force_addition is True, the appendum (inserting '-%d' etc.) should be done also for i == 0.

page_path: A function accepting an index i, the displayed page number, the number of pages, and a boolean force_addition, which creates a path to the i-th page. All arguments are as the ones for page_link.

additional_dependencies: a list of dependencies which will be added to task['uptodate']

generic_page_renderer(*lang*, *post*, *filters*, *context=None*)
Render post fragments to final HTML pages.

generic_post_list_renderer(*lang*, *posts*, *output_name*, *template_name*, *filters*, *extra_context*)
Render pages with lists of posts.

generic_renderer(*lang*, *output_name*, *template_name*, *filters*, *file_deps=None*, *uptodate_deps=None*, *context=None*, *context_deps_remove=None*, *post_deps_dict=None*, *url_type=None*, *is_fragment=False*)
Create tasks for rendering pages and post lists and other related pages.

lang is the current language. output_name is the destination file name. template_name is the template to be used. filters is the list of filters (usually site.config['FILTERS']) which will be used to post-process the result. file_deps (optional) is a list of additional file dependencies (next to template and its dependencies). uptodate_deps (optional) is a list of additional entries added to the task's uptodate list. context (optional) a dict used as a basis for the template context. The lang parameter will always be added. context_deps_remove (optional) is a list of keys to remove from the context after using it as an uptodate dependency. This should name all keys containing non-trivial Python objects; they can be replaced by adding JSON-style dicts in post_deps_dict. post_deps_dict (optional) is a dict merged into the copy of context which is used as an uptodate dependency. url_type (optional) allows to override the `URL_TYPE` configuration. is_fragment (optional) allows to write a HTML fragment instead of a HTML document.

generic_rss_feed(*lang*, *title*, *link*, *description*, *timeline*, *rss_teasers*, *rss_plain*, *feed_length=10*, *feed_url=None*, *enclosure=<function _enclosure>*, *rss_links_append_query=None*, *copyright_=None*)
Generate an ExtendedRSS2 feed object for later use.

generic_rss_renderer(*lang*, *title*, *link*, *description*, *timeline*, *output_path*, *rss_teasers*, *rss_plain*, *feed_length=10*, *feed_url=None*, *enclosure=<function _enclosure>*, *rss_links_append_query=None*, *copyright_=None*)
Take all necessary data, and render a RSS feed in output_path.

get_compiler(*source_name*)
Get the correct compiler for a post from *conf.COMPILERS*.

To make things easier for users, the mapping in conf.py is compiler->[extensions], although this is less convenient for us. The majority of this function is reversing that dictionary and error checking.

init_plugins(*commands_only=False*, *load_all=False*)
Load plugins as needed.

link(**args*, ***kwargs*)
Create a link.

parse_category_name(*category_name*)
Parse a category name into a hierarchy.

path(*kind*, *name*, *lang=None*, *is_link=False*, ***kwargs*)
Build the path to a certain kind of page.

These are mostly defined by plugins by registering via the register_path_handler method, except for slug, post_path, root and filename which are defined in this class' init method.

Here's some of the others, for historical reasons:

•root (name is ignored)

•tag_index (name is ignored)

•tag (and name is the tag name)

•tag_rss (name is the tag name)

•category (and name is the category name)

•category_rss (and name is the category name)

•archive (and name is the year, or None for the main archive index)

•index (name is the number in index-number)

•rss (name is ignored)

•gallery (name is the gallery name)

•listing (name is the source code file name)

•post_path (name is 1st element in a POSTS/PAGES tuple)

•slug (name is the slug of a post or page)

•filename (name is the source filename of a post/page, in DEFAULT_LANG, relative to conf.py)

The returned value is always a path relative to output, like "categories/whatever.html"

If is_link is True, the path is absolute and uses "/" as separator (ex: "/archive/index.html"). If is_link is False, the path is relative to output and uses the platform's separator. (ex: "archiveindex.html")

post_path(*name*, *lang*)
Link to the destination of an element in the POSTS/PAGES settings.

Example:

link://post_path/posts => /blog

register_filter(*filter_name*, *filter_definition*)
Register a filter.

filter_name should be a name not confusable with an actual executable. filter_definition should be a callable accepting one argument (the filename).

register_path_handler(*kind*, *f*)
Register a path handler.

register_shortcode(*name*, *f*)
Register function f to handle shortcode "name".

rel_link(*src*, *dst*)
Get a relative link.

render_template(*template_name*, *output_name*, *context*, *url_type=None*, *is_fragment=False*)
Render a template with the global context.

If `output_name` is None, will return a string and all URL normalization will be ignored (including the link:// scheme). If `output_name` is a string, URLs will be normalized and the resultant HTML will be saved to the named file (path must start with OUTPUT_FOLDER).

The argument `url_type` allows to override the `URL_TYPE` configuration.

If `is_fragment` is set to `True`, a HTML fragment will be rendered and not a whole HTML document.

rewrite_links(*doc*, *src*, *lang*, *url_type=None*)
Replace links in document to point to the right places.

root_path(*name*, *lang*)
Link to the current language's root.

Example:

link://root_path => /

link://root_path => /translations/spanish/

scan_posts(*really=False*, *ignore_quit=False*, *quiet=False*)
Scan all the posts.

The *quiet* option is ignored.

slug_path(*name*, *lang*)
Return a link to a post with given slug, if not ambiguous.

Example:

link://slug/yellow-camaro => /posts/cars/awful/yellow-camaro/index.html

static sort_posts_chronologically(*posts*, *lang=None*)
Sort a list of posts chronologically.

This function also takes priority, title and source path into account.

template_system

url_replacer(*src*, *dst*, *lang=None*, *url_type=None*)
Mangle URLs.

- Replaces link:// URLs with real links
- Makes dst relative to src
- Leaves fragments unchanged
- Leaves full URLs unchanged
- Avoids empty links

src is the URL where this link is used dst is the link to be mangled lang is used for language-sensitive URLs in link:// url_type is used to determine final link appearance, defaulting to URL_TYPE from config

nikola.plugin_categories module

Nikola plugin categories.

class nikola.plugin_categories.**Command**(**args*, ***kwargs*)
Bases: **:class:'nikola.plugin_categories.BasePlugin'**, **:class:'doit.cmd_base.Command'**

Doit command implementation.

cmd_options = ()

doc_description = None

doc_purpose = 'A short explanation.'

doc_usage = ''

execute (*options=None*, *args=None*)
Check if the command can run in the current environment, fail if needed, or call _execute.

name = 'dummy_command'

needs_config = True

class nikola.plugin_categories.**LateTask**
Bases: **:class:'nikola.plugin_categories.BaseTask'**

Late task generator (plugin executed after all Task plugins).

name = 'dummy_latetask'

class nikola.plugin_categories.**PageCompiler**
Bases: **:class:'nikola.plugin_categories.BasePlugin'**

Compile text files into HTML.

compile (*source*, *dest*, *is_two_file=True*, *post=None*, *lang=None*)
Compile the source file into HTML and save as dest.

compile_string (*data*, *source_path=None*, *is_two_file=True*, *post=None*, *lang=None*)
Compile the source file into HTML strings (with shortcode support).

Returns a tuple of at least two elements: HTML string [0] and shortcode dependencies [last].

config_dependencies = []

create_post (*path*, *content=None*, *onefile=False*, *is_page=False*, ***kw*)
Create post file with optional metadata.

default_metadata = {'title': '', 'date': '', 'description': '', 'category': '', 'type': 'text', 'slug': '', 'link': '', 'tags': ''}

demote_headers = False

extension ()
Return the preferred extension for the output of this compiler.

friendly_name = ''

get_compiler_extensions ()
Activate all the compiler extension plugins for a given compiler and return them.

get_dep_filename (*post*, *lang*)
Return the .dep file's name for the given post and language.

get_extra_targets (*post*, *lang*, *dest*)
Return a list of extra targets for the render_posts task when compiling the post for the specified language.

metadata_conditions = []

name = 'dummy_compiler'

read_metadata (*post*, *lang=None*)
Read the metadata from a post, and return a metadata dict.

register_extra_dependencies (*post*)
Add dependency to post object to check .dep file.

split_metadata(*data*, *post=None*, *lang=None*)
Split data from metadata in the raw post content.

supports_metadata = False

supports_onefile = True

use_dep_file = True

class nikola.plugin_categories.**RestExtension**
Bases: **:class:'nikola.plugin_categories.CompilerExtension'**

Extensions for reStructuredText.

compiler_name = 'rest'

name = 'dummy_rest_extension'

class nikola.plugin_categories.**MarkdownExtension**
Bases: **:class:'nikola.plugin_categories.CompilerExtension'**

Extensions for Markdown.

compiler_name = 'markdown'

name = 'dummy_markdown_extension'

class nikola.plugin_categories.**MetadataExtractor**
Bases: **:class:'nikola.plugin_categories.BasePlugin'**

Plugins that can extract meta information from post files.

check_requirements()
Check if requirements for an extractor are satisfied.

conditions = []

extract_filename(*filename: str*, *lang: str*) → dict
Extract metadata from filename.

extract_text(*source_text: str*) → dict
Extract metadata from text (also calls split_metadata_from_text).

map_from = None

name = 'unknown'

priority = None

requirements = []

source = None

split_metadata_from_text(*source_text: str) -> (<class 'str'>*, *<class 'str'>*)
Split text into metadata and content (both strings).

If splitting fails (there is no match), return source_text as both metadata and content. (This behavior is required for 2-file posts.)

split_metadata_re = None

supports_write = False

write_metadata(*metadata: dict*, *comment_wrap=False*) → str
Write metadata in this extractor's format.

comment_wrap is either True, False, or a 2-tuple of comments to use for wrapping, if necessary. If it's set to True, defaulting to `('<!--', '-->')` is recommended.

This function should insert comment markers (if applicable) and must insert trailing newlines.

class nikola.plugin_categories.**Task**
Bases: **:class:'nikola.plugin_categories.BaseTask'**

Task generator.

name = 'dummy_task'

class nikola.plugin_categories.**TaskMultiplier**
Bases: **:class:'nikola.plugin_categories.BasePlugin'**

Take a task and return *more* tasks.

name = 'dummy multiplier'

process(*task*)
Examine task and create more tasks. Returns extra tasks only.

class nikola.plugin_categories.**TemplateSystem**
Bases: **:class:'nikola.plugin_categories.BasePlugin'**

Provide support for templating systems.

get_deps(*filename*)
Return paths to dependencies for the template loaded from filename.

get_string_deps(*text*)
Find dependencies for a template string.

get_template_path(*template_name*)
Get the path to a template or return None.

inject_directory(*directory*)
Inject the directory with the lowest priority in the template search mechanism.

name = 'dummy_templates'

render_template(*template_name*, *output_name*, *context*)
Render template to a file using context.

This must save the data to output_name *and* return it so that the caller may do additional processing.

render_template_to_string(*template*, *context*)
Render template to a string using context.

set_directories(*directories*, *cache_folder*)
Set the list of folders where templates are located and cache.

template_deps(*template_name*)
Return filenames which are dependencies for a template.

class nikola.plugin_categories.**SignalHandler**
Bases: **:class:'nikola.plugin_categories.BasePlugin'**

Signal handlers.

name = 'dummy_signal_handler'

class nikola.plugin_categories.**ConfigPlugin**
Bases: **:class:'nikola.plugin_categories.BasePlugin'**

A plugin that can edit config (or modify the site) on-the-fly.

name = 'dummy_config_plugin'

class nikola.plugin_categories.**PostScanner**
Bases: **:class:'nikola.plugin_categories.BasePlugin'**

The scan method of these plugins is called by Nikola.scan_posts.

scan()
Create a list of posts from some source. Returns a list of Post objects.

supported_extensions()
Return a list of supported file extensions, or None if such a list isn't known beforehand.

class nikola.plugin_categories.**Taxonomy**
Bases: **:class:'nikola.plugin_categories.BasePlugin'**

Taxonomy for posts.

A taxonomy plugin allows to classify posts (see #2107) by classification strings. Classification plugins must adjust a set of options to determine certain aspects.

The following options are class attributes with their default values. These variables should be set in the class definition, in the constructor or latest in the *set_site* function.

classification_name = "taxonomy": The classification name to be used for path handlers. Must be overridden!

overview_page_items_variable_name = "items": When rendering the overview page, its template will have a list of pairs

(friendly_name, link)

for the classifications available in a variable by this name.

The template will also have a list (friendly_name, link, post_count)

for the classifications available in a variable by the name *overview_page_items_variable_name + '_with_postcount'*.

overview_page_variable_name = "taxonomy": When rendering the overview page, its template will have a list of classifications available in a variable by this name.

overview_page_hierarchy_variable_name = "taxonomy_hierarchy": When rendering the overview page, its template will have a list of tuples

(friendly_name, classification, classification_path, link, indent_levels, indent_change_before, indent_change_after)

available in a variable by this name. These tuples can be used to render the hierarchy as a tree.

The template will also have a list

(friendly_name, classification, classification_path, link, indent_levels, indent_change_before, indent_change_after, number_of_children, post_count)

available in the variable by the name *overview_page_hierarchy_variable_name + '_with_postcount'*.

more_than_one_classifications_per_post = False: If True, there can be more than one classification per post; in that case, the classification data in the metadata is stored as a list. If False, the classification data in the metadata is stored as a string, or None when no classification is given.

has_hierarchy = False: Whether the classification has a hierarchy.

include_posts_from_subhierarchies = False: If True, the post list for a classification includes all posts with a sub-classification (in case has_hierarchy is True).

include_posts_into_hierarchy_root = False: If True, include_posts_from_subhierarchies == True will also insert posts into the post list for the empty hierarchy [].

show_list_as_subcategories_list = False: If True, for every classification which has at least one subclassification, create a list of subcategories instead of a list/index of posts. This is only used when has_hierarchy = True. The template specified in subcategories_list_template will be used. If this is set to True, it is recommended to set include_posts_from_subhierarchies to True to get correct post counts.

show_list_as_index = False: Whether to show the posts for one classification as an index or as a post list.

subcategories_list_template = "taxonomy_list.tmpl": The template to use for the subcategories list when show_list_as_subcategories_list is True.

generate_atom_feeds_for_post_lists = False: Whether to generate Atom feeds for post lists in case GENERATE_ATOM is set.

template_for_single_list = "tagindex.tmpl": The template to use for the post list for one classification.

template_for_classification_overview = "list.tmpl": The template to use for the classification overview page. Set to None to avoid generating overviews.

always_disable_rss = False: Whether to always disable RSS feed generation

apply_to_posts = True: Whether this classification applies to posts.

apply_to_pages = False: Whether this classification applies to pages.

minimum_post_count_per_classification_in_overview = 1: The minimum number of posts a classification must have to be listed in the overview.

omit_empty_classifications = False: Whether post lists resp. indexes should be created for empty classifications.

add_other_languages_variable = False: In case this is *True*, each classification page will get a list of triples *(other_lang, other_classification, title)* of classifications in other languages which should be linked. The list will be stored in the variable *other_languages*.

path_handler_docstrings: A dictionary of docstrings for path handlers. See eg. nikola.py for examples. Must be overridden, keys are "taxonomy_index", "taxonomy", "taxonomy_atom", "taxonomy_rss" (but using classification_name instead of "taxonomy"). If one of the values is False, the corresponding path handler will not be created.

add_other_languages_variable = False

always_disable_rss = False

apply_to_pages = False

apply_to_posts = True

classification_name = 'taxonomy'

classify(*post*, *lang*)
Classify the given post for the given language.

Must return a list or tuple of strings.

extract_hierarchy(*classification*)
Given a classification, return a list of parts in the hierarchy.

For non-hierarchical taxonomies, it usually suffices to return *[classification]*.

generate_atom_feeds_for_post_lists = False

get_classification_friendly_name(*classification*, *lang*, *only_last_component=False*)
Extract a friendly name from the classification.

The result of this function is usually displayed to the user, instead of using the classification string.

The argument *only_last_component* is only relevant to hierarchical taxonomies. If it is set, the printable name should only describe the last component of *classification* if possible.

get_implicit_classifications(*lang*)
Return a list of classification strings which should always appear in posts_per_classification.

get_other_language_variants(*classification*, *lang*, *classifications_per_language*)
Return a list of variants of the same classification in other languages.

Given a *classification* in a language *lang*, return a list of pairs *(other_lang, other_classification)* with *lang != other_lang* such that *classification* should be linked to *other_classification*.

Classifications where links to other language versions makes no sense should simply return an empty list.

Provided is a set of classifications per language (*classifications_per_language*).

get_overview_path(*lang*, *dest_type='page'*)
Return path for classification overview.

This path handler for the classification overview must return one or two values (in this order):

- a list or tuple of strings: the path relative to OUTPUT_DIRECTORY;
- a string with values 'auto', 'always' or 'never', indicating whether INDEX_FILE should be added or not.

Note that this function must always return a list or tuple of strings; the other return value is optional with default value *'auto'*.

In case INDEX_FILE should potentially be added, the last element in the returned path must have no extension, and the PRETTY_URLS config must be ignored by this handler. The return value will be modified based on the PRETTY_URLS and INDEX_FILE settings.

dest_type can be either 'page', 'feed' (for Atom feed) or 'rss'.

get_path(*classification*, *lang*, *dest_type='page'*)
Return path to the classification page.

This path handler for the given classification must return one to three values (in this order):

- a list or tuple of strings: the path relative to OUTPUT_DIRECTORY;
- a string with values 'auto', 'always' or 'never', indicating whether INDEX_FILE should be added or not;
- an integer if a specific page of the index is to be targeted (will be ignored for post lists), or *None* if the most current page is targeted.

Note that this function must always return a list or tuple of strings; the other two return values are optional with default values *'auto'* and *None*.

In case INDEX_FILE should potentially be added, the last element in the returned path must have no extension, and the PRETTY_URLS config must be ignored by this handler. The return value will be modified based on the PRETTY_URLS and INDEX_FILE settings.

dest_type can be either 'page', 'feed' (for Atom feed) or 'rss'.

For hierarchical taxonomies, the result of extract_hierarchy is provided as *classification*. For non-hierarchical taxonomies, the classification string itself is provided as *classification*.

has_hierarchy = False

include_posts_from_subhierarchies = False

include_posts_into_hierarchy_root = False

is_enabled(*lang=None*)
Return True if this taxonomy is enabled, or False otherwise.

If lang is None, this determins whether the classification is made at all. If lang is not None, this determines whether the overview page and the classification lists are created for this language.

minimum_post_count_per_classification_in_overview = 1

more_than_one_classifications_per_post = False

name = 'dummy_taxonomy'

omit_empty_classifications = False

overview_page_hierarchy_variable_name = 'taxonomy_hierarchy'

overview_page_items_variable_name = 'items'

overview_page_variable_name = 'taxonomy'

path_handler_docstrings = {'taxonomy': '', 'taxonomy_index': '', 'taxonomy_atom': '', 'taxonomy_rss': ''}

postprocess_posts_per_classification(*posts_per_classification_per_language*, *flat_hierarchy_per_lang=None*, *hierarchy_lookup_per_lang=None*)
Rearrange, modify or otherwise use the list of posts per classification and per language.

For compatibility reasons, the list could be stored somewhere else as well.

In case *has_hierarchy* is *True*, *flat_hierarchy_per_lang* is the flat hierarchy consisting of *hierarchy_utils.TreeNode* elements, and *hierarchy_lookup_per_lang* is the corresponding hierarchy lookup mapping classification strings to *hierarchy_utils.TreeNode* objects.

provide_context_and_uptodate(*classification*, *lang*, *node=None*)
Provide data for the context and the uptodate list for the list of the given classifiation.

Must return a tuple of two dicts. The first is merged into the page's context, the second will be put into the uptodate list of all generated tasks.

For hierarchical taxonomies, node is the *hierarchy_utils.TreeNode* element corresponding to the classification.

Context must contain *title*, which should be something like 'Posts about <classification>'.

provide_overview_context_and_uptodate(*lang*)
Provide data for the context and the uptodate list for the classifiation overview.

Must return a tuple of two dicts. The first is merged into the page's context, the second will be put into the uptodate list of all generated tasks.

Context must contain *title*.

recombine_classification_from_hierarchy(*hierarchy*)
Given a list of parts in the hierarchy, return the classification string.

For non-hierarchical taxonomies, it usually suffices to return hierarchy[0].

should_generate_classification_page(*classification*, *post_list*, *lang*)
Only generates list of posts for classification if this function returns True.

should_generate_rss_for_classification_page(*classification*, *post_list*, *lang*)
Only generates RSS feed for list of posts for classification if this function returns True.

show_list_as_index = False

show_list_as_subcategories_list = False

sort_classifications (*classifications*, *lang*, *level=None*)
Sort the given list of classification strings.

Allows the plugin to order the classifications as it wants. The classifications will be ordered by *natsort* before calling this function. This function must sort in-place.

For hierarchical taxonomies, the elements of the list are a single path element of the path returned by *extract_hierarchy()*. The index of the path element in the path will be provided in *level*.

sort_posts (*posts*, *classification*, *lang*)
Sort the given list of posts.

Allows the plugin to order the posts per classification as it wants. The posts will be ordered by date (latest first) before calling this function. This function must sort in-place.

subcategories_list_template = 'taxonomy_list.tmpl'

template_for_classification_overview = 'list.tmpl'

template_for_single_list = 'tagindex.tmpl'

nikola.post module

The Post class.

class nikola.post.**Post** (*source_path*, *config*, *destination*, *use_in_feeds*, *messages*, *template_name*, *compiler*, *destination_base=None*, *metadata_extractors_by=None*)
Bases: **:class:'object'**

Represent a blog post or site page.

add_dependency (*dependency*, *add='both'*, *lang=None*)
Add a file dependency for tasks using that post.

The `dependency` should be a string specifying a path, or a callable which returns such a string or a list of strings.

The `add` parameter can be 'both', 'fragment' or 'page', to indicate that this dependency shall be used

- when rendering the fragment to HTML ('fragment' and 'both'), or
- when creating a page with parts of the `Post` embedded, which includes the HTML resulting from compiling the fragment ('page' or 'both').

If `lang` is not specified, this dependency is added for all languages.

add_dependency_uptodate (*dependency*, *is_callable=False*, *add='both'*, *lang=None*)
Add a dependency for task's `uptodate` for tasks using that post.

This can be for example an `utils.config_changed` object, or a list of such objects.

The `is_callable` parameter specifies whether `dependency` is a callable which generates an entry or a list of entries for the `uptodate` list, or whether it is an entry which can directly be added (as a single object or a list of objects).

The `add` parameter can be 'both', 'fragment' or 'page', to indicate that this dependency shall be used

- when rendering the fragment to HTML ('fragment' and 'both'), or
- when creating a page with parts of the `Post` embedded, which includes the HTML resulting from compiling the fragment ('page' or 'both').

If `lang` is not specified, this dependency is added for all languages.

Example:

post.add_dependency_uptodate(utils.config_changed({1: some_data}, 'uniqueid'), False, 'page')

alltags
Return ALL the tags for this post.

author(*lang=None*)
Return localized author or BLOG_AUTHOR if unspecified.

If lang is not specified, it defaults to the current language from templates, as set in LocaleBorg.

compile(*lang*)
Generate the cache/ file with the compiled post.

deps(*lang*)
Return a list of file dependencies to build this post's page.

deps_uptodate(*lang*)
Return a list of uptodate dependencies to build this post's page.

These dependencies should be included in `uptodate` for the task which generates the page.

description(*lang=None*)
Return localized description.

destination_path(*lang=None*, *extension='.html'*, *sep='/'*)
Destination path for this post, relative to output/.

If lang is not specified, it's the current language. Extension is used in the path if specified.

formatted_date(*date_format*, *date=None*)
Return the formatted date as unicode.

formatted_updated(*date_format*)
Return the updated date as unicode.

fragment_deps(*lang*)
Return a list of uptodate dependencies to build this post's fragment.

These dependencies should be included in `uptodate` for the task which generates the fragment.

fragment_deps_uptodate(*lang*)
Return a list of file dependencies to build this post's fragment.

guid(*lang=None*)
Return localized GUID.

has_pretty_url(*lang*)
Check if this page has a pretty URL.

hyphenate

is_mathjax
Return True if this post has the mathjax tag in the current language or is a python notebook.

is_translation_available(*lang*)
Return True if the translation actually exists.

next_post
Return next post.

paragraph_count
Return the paragraph count for this post.

permalink(*lang=None*, *absolute=False*, *extension='.html'*, *query=None*)
Return permalink for a post.

prev_post
Return previous post.

previewimage
Return the previewimage path.

reading_time
Return reading time based on length of text.

register_depfile(*dep*, *dest=None*, *lang=None*)
Register a dependency in the dependency file.

remaining_paragraph_count
Return the remaining paragraph count for this post (does not include teaser).

remaining_reading_time
Remaining reading time based on length of text (does not include teaser).

section_color(*lang=None*)
Return the color of the post's section.

section_link(*lang=None*)
Return the link to the post's section (deprecated).

section_name(*lang=None*)
Return the name of the post's section.

section_slug(*lang=None*)
Return the slug for the post's section.

source_ext(*prefix=False*)
Return the source file extension.

If *prefix* is True, a *.src.* prefix will be added to the resulting extension if it's equal to the destination extension.

source_link(*lang=None*)
Return absolute link to the post's source.

tags
Return tags for the current language.

tags_for_language(*lang*)
Return tags for a given language.

template_name
Return template name for this post.

text(*lang=None*, *teaser_only=False*, *strip_html=False*, *show_read_more_link=True*, *feed_read_more_link=False*, *feed_links_append_query=None*)
Read the post file for that language and return its contents.

teaser_only=True breaks at the teaser marker and returns only the teaser. strip_html=True removes HTML tags show_read_more_link=False does not add the Read more... link feed_read_more_link=True uses FEED_READ_MORE_LINK instead of INDEX_READ_MORE_LINK lang=None uses the last used to set locale

All links in the returned HTML will be relative. The HTML returned is a bare fragment, not a full document.

title(*lang=None*)
Return localized title.

If lang is not specified, it defaults to the current language from templates, as set in LocaleBorg.

translated_base_path(*lang*)
Return path to the translation's base_path file.

translated_source_path(*lang*)
Return path to the translation's source file.

static write_depfile(*dest*, *deps_list*, *post=None*, *lang=None*)
Write a depfile for a given language.

nikola.rc4 module

nikola.shortcodes module

Support for Hugo-style shortcodes.

exception nikola.shortcodes.**ParsingError**
Bases: **:class:'Exception'**

Used for forwarding parsing error messages to apply_shortcodes.

nikola.shortcodes.**apply_shortcodes**(*data*, *registry*, *site=None*, *filename=None*, *raise_exceptions=False*, *lang=None*, *extra_context=None*)
Apply Hugo-style shortcodes on data.

{{% name parameters %}} will end up calling the registered "name" function with the given parameters. {{% name parameters %}} something {{% /name %}} will call name with the parameters and one extra "data" parameter containing " something ".

If raise_exceptions is set to True, instead of printing error messages and terminating, errors are passed on as exceptions to the caller.

The site parameter is passed with the same name to the shortcodes so they can access Nikola state.

```
>>> print(apply_shortcodes('==> {{% foo bar=baz %}} <==', {'foo': lambda *a, **k:␣
↪k['bar']}))
==> baz <==
>>> print(apply_shortcodes('==> {{% foo bar=baz %}}some data{{% /foo %}} <==', {
↪'foo': lambda *a, **k: k['bar']+k['data']}))
==> bazsome data <==
```

nikola.shortcodes.**extract_shortcodes**(*data*)
Return data with replaced shortcodes, shortcodes.

data is the original data, with the shortcodes replaced by UUIDs.

a dictionary of shortcodes, where the keys are UUIDs and the values are the shortcodes themselves ready to process.

nikola.state module

Persistent state implementation.

class nikola.state.**Persistor**(*path*)
Bases: **:class:'object'**

Persist stuff in a place.

This is an intentionally dumb implementation. It is *not* meant to be fast, or useful for arbitrarily large data. Use lightly.

Intentionally it has no namespaces, sections, etc. Use as a responsible adult.

delete(*key*)
Delete key and the value it contains.

get(*key*)
Get data stored in key.

set(*key*, *value*)
Store value in key.

nikola.utils module

Utility functions.

class nikola.utils.**CustomEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Bases: **:class:'json.encoder.JSONEncoder'**

Custom JSON encoder.

default(*obj*)
Create default encoding handler.

nikola.utils.**get_theme_path**(*theme*)
Return the theme's path, which equals the theme's name.

nikola.utils.**get_theme_path_real**(*theme*, *themes_dirs*)
Return the path where the given theme's files are located.

Looks in ./themes and in the place where themes go when installed.

nikola.utils.**get_theme_chain**(*theme*, *themes_dirs*)
Create the full theme inheritance chain including paths.

nikola.utils.**load_messages**(*themes*, *translations*, *default_lang*, *themes_dirs*)
Load theme's messages into context.

All the messages from parent themes are loaded, and "younger" themes have priority.

nikola.utils.**copy_tree**(*src*, *dst*, *link_cutoff=None*, *ignored_filenames=None*)
Copy a src tree to the dst folder.

Example:

src = "themes/default/assets" dst = "output/assets"

should copy "themes/defauts/assets/foo/bar" to "output/assets/foo/bar"

If link_cutoff is set, then the links pointing at things *inside* that folder will stay as links, and links pointing *outside* that folder will be copied.

ignored_filenames is a set of file names that will be ignored.

nikola.utils.**copy_file**(*source*, *dest*, *cutoff=None*)
Copy a file from source to dest. If link target starts with *cutoff*, symlinks are used.

nikola.utils.**slugify**(*value*, *lang=None*, *force=False*)
Normalize string, convert to lowercase, remove non-alpha characters, convert spaces to hyphens.

From Django's "django/template/defaultfilters.py".

```
>>> print(slugify('ääí.óú', lang='en'))
aeiou
```

```
>>> print(slugify('foo/bar', lang='en'))
foobar
```

```
>>> print(slugify('foo bar', lang='en'))
foo-bar
```

nikola.utils.**unslugify**(*value*, *lang=None*, *discard_numbers=True*)
Given a slug string (as a filename), return a human readable string.

If discard_numbers is True, numbers right at the beginning of input will be removed.

nikola.utils.**to_datetime**(*value*, *tzinfo=None*)
Convert string to datetime.

nikola.utils.**apply_filters**(*task*, *filters*, *skip_ext=None*)
Apply filters to a task.

If any of the targets of the given task has a filter that matches, adds the filter commands to the commands of the task, and the filter itself to the uptodate of the task.

class nikola.utils.**config_changed**(*config*, *identifier=None*)
Bases: **:class:'doit.tools.config_changed'**

A copy of doit's config_changed, using pickle instead of serializing manually.

configure_task(*task*)
Configure a task with a digest.

nikola.utils.**get_crumbs**(*path*, *is_file=False*, *index_folder=None*, *lang=None*)
Create proper links for a crumb bar.

index_folder is used if you want to use title from index file instead of folder name as breadcrumb text.

```
>>> crumbs = get_crumbs('galleries')
>>> len(crumbs)
1
>>> crumbs[0]
['#', 'galleries']
```

```
>>> crumbs = get_crumbs(os.path.join('galleries','demo'))
>>> len(crumbs)
2
>>> crumbs[0]
['..', 'galleries']
>>> crumbs[1]
['#', 'demo']
```

```
>>> crumbs = get_crumbs(os.path.join('listings','foo','bar'), is_file=True)
>>> len(crumbs)
3
>>> crumbs[0]
['..', 'listings']
>>> crumbs[1]
['.', 'foo']
```

```
>>> crumbs[2]
['#', 'bar']
```

nikola.utils.**get_tzname**(*dt*)
Given a datetime value, find the name of the time zone.

DEPRECATED: This thing returned basically the 1st random zone that matched the offset.

nikola.utils.**get_asset_path**(*path*, *themes*, *files_folders={'files': ''}*, *output_dir='output'*)
Return the "real", absolute path to the asset.

By default, it checks which theme provides the asset. If the asset is not provided by a theme, then it will be checked for in the FILES_FOLDERS. If it's not provided by either, it will be chacked in output, where it may have been created by another plugin.

```
>>> print(get_asset_path('assets/css/nikola_rst.css', get_theme_chain('bootstrap3
↪', ['themes'])))
/.../nikola/data/themes/base/assets/css/nikola_rst.css
```

```
>>> print(get_asset_path('assets/css/theme.css', get_theme_chain('bootstrap3', [
↪'themes'])))
/.../nikola/data/themes/bootstrap3/assets/css/theme.css
```

```
>>> print(get_asset_path('nikola.py', get_theme_chain('bootstrap3', ['themes']), {
↪'nikola': ''}))
/.../nikola/nikola.py
```

```
>>> print(get_asset_path('nikola.py', get_theme_chain('bootstrap3', ['themes']), {
↪'nikola': 'nikola'}))
None
```

```
>>> print(get_asset_path('nikola/nikola.py', get_theme_chain('bootstrap3', [
↪'themes']), {'nikola': 'nikola'}))
/.../nikola/nikola.py
```

nikola.utils.**unicode_str**
alias of **:class:'str'**

nikola.utils.**bytes_str**
alias of **:class:'bytes'**

nikola.utils.**unichr**()
Return a Unicode string of one character with ordinal i; 0 <= i <= 0x10ffff.

class nikola.utils.**Functionary**(*default*, *default_lang*)
Bases: **:class:'collections.defaultdict'**

Class that looks like a function, but is a defaultdict.

class nikola.utils.**TranslatableSetting**(*name*, *inp*, *translations*)
Bases: **:class:'object'**

A setting that can be translated.

You can access it via: SETTING(lang). You can omit lang, in which case Nikola will ask LocaleBorg, unless you set SETTING.lang, which overrides that call.

You can also stringify the setting and you will get something sensible (in what LocaleBorg claims the language is, can also be overriden by SETTING.lang). Note that this second method is deprecated. It is kept for backwards compatibility and safety. It is not guaranteed.

The underlying structure is a defaultdict. The language that is the default value of the dict is provided with __init__().

default_lang = 'en'

format(**args*, ***kwargs*)
Format ALL the values in the setting the same way.

get_lang()
Return the language that should be used to retrieve settings.

lang = None

langformat(*formats*)
Format ALL the values in the setting, on a per-language basis.

class nikola.utils.**TemplateHookRegistry**(*name*, *site*)
Bases: **:class:'object'**

A registry for template hooks.

Usage:

```
>>> r = TemplateHookRegistry('foo', None)
>>> r.append('Hello!')
>>> r.append(lambda x: 'Hello ' + x + '!', False, 'world')
>>> repr(r())
'Hello!\nHello world!'
```

append(*inp*, *wants_site_and_context=False*, **args*, ***kwargs*)
Register an item.

inp can be a string or a callable returning one. *wants_site* tells whether there should be a *site* keyword

argument provided, for accessing the site.

Further positional and keyword arguments are passed as-is to the callable.

wants_site, args and kwargs are ignored (but saved!) if *inp* is not callable. Callability of *inp* is determined only once.

calculate_deps()
Calculate dependencies for a registry.

generate()
Generate items.

class nikola.utils.**LocaleBorg**
Bases: **:class:'object'**

Provide locale related services and autoritative current_lang.

current_lang is the last lang for which the locale was set and is meant to be set only by LocaleBorg.set_locale.

python's locale code should not be directly called from code outside of LocaleBorg, they are compatibilty issues with py version and OS support better handled at one central point, LocaleBorg.

In particular, don't call locale.setlocale outside of LocaleBorg.

Assumptions: We need locales only for the languages there is a nikola translation. We don't need to support current_lang through nested contexts

Usage: # early in cmd or test execution LocaleBorg.initialize(...)

any time later lang = LocaleBorg().<service>

Available services: .current_lang : autoritative current_lang , the last seen in set_locale .set_locale(lang) : sets current_lang and sets the locale for lang .get_month_name(month_no, lang) : returns the localized month name

NOTE: never use locale.getlocale() , it can return values that locale.setlocale will not accept in Windows XP, 7 and pythons 2.6, 2.7, 3.3 Examples: "Spanish", "French" can't do the full circle set / get / set

classmethod add_handler(*month_name_handler=None*, *formatted_date_handler=None*)
Allow to add month name and formatted date handlers.

If month_name_handler is not None, it is expected to be a callable which accepts (month_no, lang) and returns either a string or None.

If formatted_date_handler is not None, it is expected to be a callable which accepts (date_format, date, lang) and returns either a string or None.

A handler is expected to either return the correct result for the given language and data, or return None to indicate it is not able to do the job. In that case, the next handler is asked, and finally the default implementation is used.

current_lang
Return the current language.

formatted_date(*date_format*, *date*)
Return the formatted date as unicode.

get_month_name(*month_no*, *lang*)
Return localized month name in an unicode string.

classmethod initialize(*locales*, *initial_lang*)
Initialize LocaleBorg.

locales [dict with lang: locale_n] the same keys as in nikola's TRANSLATIONS locale_n a sanitized locale, meaning

locale.setlocale(locale.LC_ALL, locale_n) will succeed locale_n expressed in the string form, like "en.utf8"

initialized = False

classmethod reset()
Reset LocaleBorg.

Used in testing to prevent leaking state between tests.

set_locale(*lang*)
Set the locale for language lang, returns an empty string.

in linux the locale encoding is set to utf8, in windows that cannot be guaranted. In either case, the locale encoding is available in cls.encodings[lang]

nikola.utils.**sys_encode**(*thing*)
Return bytes encoded in the system's encoding.

nikola.utils.**sys_decode**(*thing*)
Return Unicode.

nikola.utils.**makedirs**(*path*)
Create a folder and its parents if needed (mkdir -p).

nikola.utils.**get_parent_theme_name**(*theme_name*, *themes_dirs=None*)
Get name of parent theme.

nikola.utils.**demote_headers**(*doc*, *level=1*)
Demote <hN> elements by one.

nikola.utils.**get_translation_candidate**(*config*, *path*, *lang*)
Return a possible path where we can find the translated version of some page, based on the TRANSLATIONS_PATTERN configuration variable.

```
>>> config = {'TRANSLATIONS_PATTERN': '{path}.{lang}.{ext}', 'DEFAULT_LANG': 'en',
↪ 'TRANSLATIONS': {'es':'1', 'en': 1}}
>>> print(get_translation_candidate(config, '*.rst', 'es'))
*.es.rst
>>> print(get_translation_candidate(config, 'fancy.post.rst', 'es'))
fancy.post.es.rst
>>> print(get_translation_candidate(config, '*.es.rst', 'es'))
*.es.rst
>>> print(get_translation_candidate(config, '*.es.rst', 'en'))
*.rst
>>> print(get_translation_candidate(config, 'cache/posts/fancy.post.es.html', 'en
↪'))
cache/posts/fancy.post.html
>>> print(get_translation_candidate(config, 'cache/posts/fancy.post.html', 'es'))
cache/posts/fancy.post.es.html
>>> print(get_translation_candidate(config, 'cache/pages/charts.html', 'es'))
cache/pages/charts.es.html
>>> print(get_translation_candidate(config, 'cache/pages/charts.html', 'en'))
cache/pages/charts.html
```

```
>>> config = {'TRANSLATIONS_PATTERN': '{path}.{ext}.{lang}', 'DEFAULT_LANG': 'en',
↪ 'TRANSLATIONS': {'es':'1', 'en': 1}}
>>> print(get_translation_candidate(config, '*.rst', 'es'))
*.rst.es
>>> print(get_translation_candidate(config, '*.rst.es', 'es'))
*.rst.es
>>> print(get_translation_candidate(config, '*.rst.es', 'en'))
*.rst
>>> print(get_translation_candidate(config, 'cache/posts/fancy.post.html.es', 'en
↪'))
cache/posts/fancy.post.html
>>> print(get_translation_candidate(config, 'cache/posts/fancy.post.html', 'es'))
cache/posts/fancy.post.html.es
```

nikola.utils.**write_metadata**(*data*, *metadata_format=None*, *comment_wrap=False*, *site=None*, *compiler=None*)
Write metadata.

Recommended usage: pass *site*, *comment_wrap* (True, False, or a 2-tuple of start/end markers), and optionally *compiler*. Other options are for backwards compatibility.

nikola.utils.**ask**(*query*, *default=None*)
Ask a question.

nikola.utils.**ask_yesno**(*query*, *default=None*)
Ask a yes/no question.

nikola.utils.**options2docstring**(*name*, *options*)
Translate options to a docstring.

nikola.utils.**os_path_split**(*path*)
Split a path.

nikola.utils.**get_displayed_page_number**(*i*, *num_pages*, *site*)
Get page number to be displayed for entry *i*.

nikola.utils.**adjust_name_for_index_path_list**(*path_list*, *i*, *displayed_i*, *lang*, *site*, *force_addition=False*, *extension=None*)
Retrurn a path list for a given index page.

nikola.utils.**adjust_name_for_index_path**(*name*, *i*, *displayed_i*, *lang*, *site*, *force_addition=False*, *extension=None*)
Return file name for a given index file.

nikola.utils.**adjust_name_for_index_link**(*name*, *i*, *displayed_i*, *lang*, *site*, *force_addition=False*, *extension=None*)
Return link for a given index file.

class nikola.utils.**NikolaPygmentsHTML**(*anchor_ref*, *classes=None*, *linenos='table'*, *linenostart=1*)
Bases: **:class:'pygments.formatters.html.HtmlFormatter'**

A Nikola-specific modification of Pygments' HtmlFormatter.

wrap(*source*, *outfile*)
Wrap the `source`, which is a generator yielding individual lines, in custom generators.

nikola.utils.**create_redirect**(*src*, *dst*)
Create a redirection.

nikola.utils.**clean_before_deployment**(*site*)
Clean drafts and future posts before deployment.

nikola.utils.**sort_posts**(*posts*, **keys*)
Sort posts by a given predicate. Helper function for templates.

If a key starts with '-', it is sorted in descending order.

Usage examples:

```
sort_posts(timeline, 'title', 'date')
sort_posts(timeline, 'author', '-section_name')
```

nikola.utils.**indent**(*text*, *prefix*, *predicate=None*)
Add 'prefix' to the beginning of selected lines in 'text'.

If 'predicate' is provided, 'prefix' will only be added to the lines where 'predicate(line)' is True. If 'predicate' is not provided, it will default to adding 'prefix' to all non-empty lines that do not consist solely of whitespace characters.

nikola.utils.**load_data**(*path*)
Given path to a file, load data from it.

nikola.utils.**html_unescape**(*s*)
Convert all named and numeric character references (e.g. >, >, &x3e;) in the string s to the corresponding unicode characters. This function uses the rules defined by the HTML 5 standard for both valid and invalid character references, and the list of HTML 5 named character references defined in html.entities.html5.

nikola.utils.**rss_writer**(*rss_obj*, *output_path*)
Write an RSS object to an xml file.

nikola.utils.**map_metadata**(*meta*, *key*, *config*)
Map metadata from other platforms to Nikola names.

This uses the METADATA_MAPPING setting (via `config`) and modifies the dict in place.

class nikola.utils.**TreeNode**(*name*, *parent=None*)
Bases: **:class:'object'**

A tree node.

get_children()
Get children of a node.

get_path()
Get path.

indent_change_after = 0

indent_change_before = 0

indent_levels = None

nikola.utils.**clone_treenode**(*treenode*, *parent=None*, *acceptor=<function <lambda>>*)
Clone a TreeNode.

Children are only cloned if *acceptor* returns *True* when applied on them.

Returns the cloned node if it has children or if *acceptor* applied to it returns *True*. In case neither applies, *None* is returned.

nikola.utils.**flatten_tree_structure**(*root_list*)
Flatten a tree.

nikola.utils.**sort_classifications**(*taxonomy*, *classifications*, *lang*)
Sort the given list of classifications of the given taxonomy and language.

`taxonomy` must be a `Taxonomy` plugin. `classifications` must be an iterable collection of classification strings for that taxonomy. `lang` is the language the classifications are for.

The result will be returned as a sorted list. Sorting will happen according to the way the complete classification hierarchy for the taxonomy is sorted.

nikola.utils.**join_hierarchical_category_path**(*category_path*)
Join a category path.

nikola.utils.**parse_escaped_hierarchical_category_name**(*category_name*)
Parse a category name.

nikola.winutils module

windows utilities to workaround problems with symlinks in a git clone.

nikola.winutils.**fix_all_git_symlinked**(*topdir*)
Convert git symlinks to real content.

Most (all?) of git implementations in windows store a symlink pointing into the repo as a text file, the text being the relative path to the file with the real content.

So, in a clone of nikola in windows the symlinked files will have the wrong content; a .zip download from Github has the same problem.

This function will rewrite each symlinked file with the correct contents, but keep in mind that the working copy will be seen as dirty by git after operation.

Expects to find a list of symlinked files at nikola/data/symlinked.txt

The list can be generated by scripts/generate_symlinked_list.sh , which is basically a redirect of

```
cd nikola_checkout git ls-files -s | awk '/120000/{print $4}'
```

Weakness: if interrupted of fail amidst a directory copy, next run will not see the missing files.

nikola.winutils.**is_file_into_dir**(*filename*, *dirname*)
Check if a file is in directory.

Module contents

Nikola – a modular, fast, simple, static website generator.

Python Module Index

n

Index

A

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y