Soho Documentation

Release 0.8.0

Damien Baty

Sep 27, 2017

Contents

Soho is a static web site generator. Yet another one? Yes. And I am afraid it may not be more extensible, faster or in any way better than the multitude of other similar tools. I wrote a first version of it a few years ago (when there were less choice and none fit my needs) and wanted to continue using the same source files for a couple of my web sites.

- Soho is small. It is only a few hundred lines long and can easily be read, evaluated and tweaked.
- Soho currently supports reStructuredText and Page Templates (a.k.a. Zope Page Templates) and obviously HTML but it may be extended to support other markup languages (e.g. Markdown) as well as other templating systems (e.g. Jinja).
- Soho can generate web sites in multiple languages and supports I18N (internationalization) through GNU gettext files. For further details, see the related section in the *Tutorial*.
- Soho does **not** automatically generate a blog-like archive site structure, or anything usually needed by blogs (such as tags or syndication feeds). If you need a blog generator, you may want to look at other tools (Pelican seems to be a popular choice).

CHAPTER 1

Topics

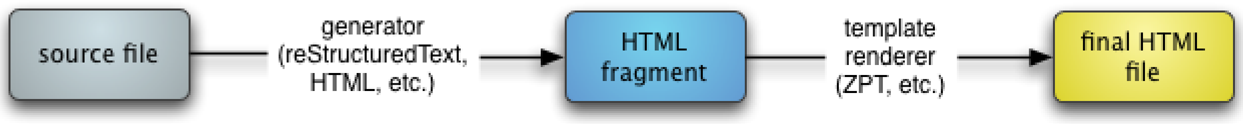
Basics

Soho consists of a single command-line program called `soho-build`. Running it on a specially structured directory will generate HTML files, copy assets (such as images, CSS or JavaScript files) and generate a site map.

For each file to process (thus excluding assets), the following tasks are done:

1. Generate HTML from the source file. The source file may be written in a special markup such as reStructuredText or directly in HTML. In all cases, a generator is chosen based on the file extension and returns an HTML fragment.
2. Apply a template to this HTML fragment. Currently, only Page Templates (also known as Zope Page Templates or ZPT) is supported, but Soho can be extended to support other templating systems. When applying the template, additional bindings can be provided through a metadata file (or directly in the source file if possible, e.g. if the source file is in reStructuredText).

All in all, here is a diagram of the process:


source file
generator (reStructuredText, HTML, etc.)
HTML fragment
template renderer (ZPT, etc.)
final HTML file

Installation

You must have Python 2.7 or Python 3.2 installed. Other versions may work but are not supported. As usual, it is recommended that you install Soho within a virtual environment (virtualenv).

Soho and required dependencies can be installed like this (of course, you may use `pip` instead of `easy_install`):

```
$ easy_install Soho
```

Required dependencies are kept to a minimum. In fact, as of this writing, there are no required dependency at all. Depending on the features that you would like to use, you need to install the corresponding extra requirements.

To use Page Templates:

```
$ easy_install Soho[zpt]
```

To use the reStructuredText generator:

```
$ easy_install Soho[rst]
```

If you are used to write documentation with Sphinx and the reStructuredText directives it defines (e.g. `code-block`), you may want to install Sphinx as well.

To benefit from I18N support:

```
$ easy_install Soho[i18n]
```

If you are feeling adventurous, you may install all optional dependencies with the following command:

```
$ easy_install Soho[all]
```

To check that Soho has been correctly installed, you may look for a new `soho-build` executable file that should have been added to your `PATH`:

```
$ soho-build --version
soho-build 0.8.0
```

Tutorial

The goal of this tutorial is to show most features of Soho, step by step. We will first start with the basics, add assets, then show how to use metadata and pass bindings to the templates, and conclude with internationalization. All sections (except the last one about internationalization) use the same set of files that are progressively updated. The resources (source and configuration files) of each section of this tutorial can be found in the docs/_tutorial folder in the source.

Setting up the structure and creating source files

Let's create a new directory that will contain our site: the source files, assets (images, Javascript, CSS, etc.) and a Soho configuration file.

```
$ mkdir site1
$ cd site1
$ export SITE_ROOT=`pwd`
```

The last command is just a way to get an "anchor" to the root of your site structure. It is not used by Soho at all but may prove useful in the rest of this tutorial.

We need a directory for the source files. By default, Soho looks for a directory named `src` so that is what we will use here.

```
$ mkdir src
$ cd src
```

We can already create our first source file there named `index.rst`, with the following content:

```
This is the **home** page. Look, here is a `link`_ to the second page.

.. _link: second.html
```

As you can see, there is a link to another file here, called `second.html`, which we can create with the following content:

```
<p>
  You have reached the second page. Wouldn't you like to go back to
  the <a href="index.html">home page</a>?
</p>
```

Now that we have our content, we need a template. By default, templates are looked up in a `templates` directory next to the `src` directory (that we already have).

```
$ cd $SITE_ROOT
$ mkdir templates
$ ls
src     templates
$ cd templates
```

We are going to create our first template named `layout.pt` in this new `templates` directory. We will start with a very simple template indeed:

```
<div class="content" tal:content="structure body"/>
<div class="footer">Generated by Soho.</div>
```

We are using Page Templates (also known as Zope Page Templates or ZPT). As you can notice, the only binding here is named `body` and contain the HTML fragment that has been generated from our source files. We will see later that we can pass additional bindings.

This is enough for a first test. We just need to add a configuration file next to the `src` and `templates` directory, which we will name `sohoconf.py`. The configuration file is a regular Python file:

```
asset_dir = None
locale_dir = None
template = 'layout.pt'
```

Here we need to disable some default values that correspond to features that will be covered later. Let's check our files before running Soho for the first time:

```
$ cd $SITE_ROOT
$ ls
src   sohoconf.py     templates
```

We are ready to run:

```
$ soho-build
<date> - INFO - Building HTML files...
<date> - INFO - Processing "/path/to/site1/src/index.rst" (writing in "/path/to/site1/
↪www/index.html").
<date> - INFO - Processing "/path/to/site1/src/second.html" (writing in "/path/to/
↪site1/www/second.html").
<date> - INFO - Generating Sitemap...
<date> - INFO - Done.
$ ls www
index.html   second.html     sitemap.xml
```

As you can see, our source files have been transformed to HTML, the template has been applied, and the resulting web site has been saved in files in a new `www` directory. Also, Soho generated a Sitemap.

Note: If you look closely at the generated `sitemap.xml`, you may see that the URLs all have some kind of default prefix (`http://exemple.com/soho/default-base-url`). You may change this prefix by setting the `base_url` variable in the configuration file (if you do so, be sure to run `soho-build` with the `--force` command-line option).

You may continue with the tutorial by following the *Adding assets* section.

Adding assets

(This is a continuation of the first section of the tutorial entitled *Setting up the structure and creating source files*.)

Our web site is fine, although not visually appealing. As specialists put it, it is ugly. We are going to use a CSS file to "enhance the user experience" (i.e. make it look a bit better). We could put this CSS file in the `src` directory, but it is not content per se. So we will put it in a special directory named `assets` or, rather, in a sub-directory called `assets/css`.

```
$ cd $SITE_ROOT
$ mkdir -p assets/css
$ ls
assets    sohoconf.py    src    templates
$ cd assets/css
```

Here is a very simple CSS file (which you may obviously improve):

```
.footer {
    border-top: 1px solid black;
    margin-top: 1em;
    padding-top: 1em;
}
```

We need to indicate to Soho that we have a special directory for assets. This is done in the `sohoconf.py` file with the `asset_dir` variable. You may remember that we set it to `None` in the first section of the tutorial to indicate that we did not have any asset directory. Here, we have different solutions to indicate the directory. This is the same mechanism for all path settings in Soho. We may indicate an absolute path like this:

```
asset_dir = '/path/to/your/site1/assets'
```

A usually better solution is to indicate a path relative to the configuration file, like this (note the leading `./`):

```
asset_dir = './assets'
```

In fact, this is the default value for `asset_dir`, so you do not have to provide it. Hence, the complete configuration file could look like this:

```
locale_dir = None
template = 'layout.pt'
```

We also need to add a link to this stylesheet in the template (`templates/layout.pt`). Because we are good citizens, we will write a valid HTML document:

```
<!DOCTYPE html>
<html>
```

```
<head>
  <title>My first web site with Soho</title>
  <link rel="stylesheet" type="text/css" href="css/style.css"/>
</head>
<body>
<div class="content" tal:content="structure body"/>
<div class="footer">Generated by Soho.</div>
</body>
</html>
```

We are ready to run Soho again:

```
$ cd $SITE_ROOT
$ ls
assets    sohoconf.py    src    templates
$ soho-build
<date> - INFO - Copying assets...
<date> - INFO - Copying "/path/to/site1/assets/css/style.css" to "/path/to/site1/www/
↪css/style.css"
<date> - INFO - Building HTML files...
<date> - INFO - Done.
```

As you can see, only assets have been processed. This is because we did not change any source file. However, we did change the template, so we should force the processing of source files. This can be done with the `-f` (or `--force`) command-line option, like this:

```
$ soho-build -f
<date> - INFO - Copying assets...
<date> - INFO - Copying "/path/to/site1/assets/css/style.css" to "/path/to/site1/www/
↪css/style.css"
<date> - INFO - Building HTML files...
<date> - INFO - Processing "/path/to/site1/src/index.rst" (writing in "/path/to/site1/
↪www/index.html").
<date> - INFO - Processing "/path/to/site1/src/second.html" (writing in "/path/to/
↪site1/www/second.html").
<date> - INFO - Generating Sitemap...
<date> - INFO - Done.
```

You may continue with the tutorial by following the *Metadata and template bindings* section.

Metadata and template bindings

(This is a continuation of the second section of the tutorial entitled *Adding assets*.)

You may have seen that since the last iteration of the template, we now have a `<title>` tag. It would be a good thing to have it filled with the title of each document. For this, additional bindings should be passed to the template. This can be done by providing metadata about each file (and even directories).

We will start by setting metadata on `second.html`. Metadata are looked up in `*.meta.py` files. Hence, we are going to create a `second.html.meta.py` file next to `second.html`, with the following content:

```
title = 'The second page'
```

Metadata files are Python files so you may import modules, compute things, etc. All symbols that are local to the file will be available in a binding called `md` in the template.

We could hence change the template like this (changes appear on the highlighted line):

```
<!DOCTYPE html>
<html>
<head>
  <title tal:content="md['title']"/>
  <link rel="stylesheet" type="text/css" href="css/style.css"/>
</head>
<body>
<div class="content" tal:content="structure body"/>
<div class="footer">Generated by Soho.</div>
</body>
</html>
```

We did set metadata for the `second.html` file but not for `index.rst`. We could create an `index.rst.meta.py` like we did above, but reStructuredText files can embed metadata. We will indicate the title in the file itself, like this (see highlighted lines):

```
.. meta::
   :title: The home page

This is the **home** page. Look, here is a `link`_ to the second page.

.. _link: second.html
```

We are ready to run Soho again:

```
$ cd $SITE_ROOT
$ ls src
index.rst     second.html    second.html.meta.py
$ soho-build -f
[...]
```

You should now see the title of each page in your web browser.

In fact, metadata can be set on directories in files named `.meta.py`. The metadata of each file automatically inherits from the metadata of the directory it lives in, as well as the directory above (if any), and so on and so forth. We will see a use-case for this in the next section entitled *Internationalization and metadata on directory*.

Internationalization and metadata on directory

Here, we will build a web site with a more complex structure and content in two languages: English and French. We will start from scratch with a new empty directory:

```
$ mkdir site2
$ cd site2
$ export SITE_ROOT=`pwd`
```

Again, the `SITE_ROOT` environment variable is not used by Soho but will be useful to keep track of directory changes below.

The web site will have two sections, one for each language:

```
$ mkdir src
$ cd src
$ mkdir en
$ mkdir fr
```

Once we have that, we may proceed and add content with two source files in each section. First, `src/en/index.html`:

```
This is the home page.
```

Then `src/en/contact.html`:

```
You may indeed contact me.
```

And a home page in French in `src/fr/index.html`:

```
Ceci est une page d'accueil.
```

Finally, the contact page in `src/fr/contact.html`:

```
Vous pouvez me contacter.
```

Until now, this is very similar to what we have done in the previous section of the tutorial. We have translated content in source files. We will now add some automatically translated content in the template (`templates/layout.pt`):

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title tal:content="md['title']"/>
5    <link rel="stylesheet" type="text/css" href="/css/style.css"/>
6  </head>
7  <body i18n:domain="tutorial">
8  <div class="content" tal:content="structure body"/>
9  <div class="footer">
10   <a href="contact.html" i18n:translate="">Contact me!</a>
11 </div>
12 </body>
13 </html>
```

This is the same template as in the previous section except for the highlighted lines. As you can see, we set the I18N domain at line 7. We also define a link whose text will be automatically translated at line 10.

The template uses a CSS file, so you may want to copy the CSS file of the previous section to a new `assets/css` directory.

Now we must set the translation somewhere. The default text ("Contact me") is in English, so we only need to define the translation in French. This must be done in a file that follows the GNU gettext format. We could write this file from scratch (the format is simple enough), but we will use Python tools to help us instead:

- Babel, which provides tools that will help us generate the translation files;
- Lingua, which provides message extractors. In other words, Lingua detects which messages have to be translated.

Warning: As of this writing, Babel and Lingua support of Python 3 is unknown. See the *note below* for an alternative.

```
$ easy_install Babel lingua
```

(As usual, you may use `pip install` instead of `easy_install`.)

For Babel to work, we can use a dummy `setup.py` file in the root directory of our web site (at the same level as `src`). Such files are usually written to describe and create Python packages. Here, we will only use the portion that is needed by Babel.

```
from setuptools import setup


# Of course, this is not a real Python package. We use 'setup()' only
# to register message extractors.
setup(packages=(),
      message_extractors={'.': (
            ('src/**meta.py', 'lingua_python', None),
            ('templates/**.pt', 'lingua_xml', None),
            )}
      )
```

These few lines instruct Babel to use Lingua to extract messages from templates (`*.pt` files in `templates`) and metadata files (`*.meta.py`). As you can see, this file is very simple and can be reused for any Soho web site.

We also need to tell Babel where we want the translation files to be stored. We can do so in a `setup.cfg` file.

```
[init_catalog]
domain = tutorial
input_file = locale/tutorial.pot
output_dir = locale

[extract_messages]
output_file = locale/tutorial.pot
width = 80

[update_catalog]
domain = tutorial
input_file = locale/tutorial.pot
output_dir = locale
previous = true

[compile_catalog]
domain = tutorial
directory = locale
statistics = true
```

This file tells Babel about the i18n domain that we want to handle (`tutorial`), the path where files will be created (a `locale` directory, which is the standard name and the default in Soho) and a few other settings that you may read about in Babel documentation. We will create this directory and get ready to generate our translation files:

```
$ cd $SITE_ROOT
$ mkdir locale
$ ls
assets    locale    setup.cfg    setup.py    sohoconf.py    src    templates    www
```

First, we extract messages from our template with the `extract_messages` command:

```
$ python setup.py extract_messages
running extract_messages
extracting messages from templates/layout.pt
writing PO template file to locale/tutorial.pot
$ ls locale
tutorial.pot
```

This `tutorial.pot` file is a template. It contains a preamble and a list of messages to be translated:

```
$ tail -n 4 locale/tutorial.pot
#: templates/layout.pt:10
msgid "Contact me!"
msgstr ""
```

As indicated above, our message is already in English, so we need only a French translation that we will create with the `init_catalog` command:

```
$ python setup.py init_catalog -l fr
running init_catalog
creating catalog 'locale/fr/LC_MESSAGES/tutorial.po' based on 'locale/tutorial.pot'
```

A "catalog" has been created in the indicated path and it will look very much like the `tutorial.pot` template above. The preamble will be slightly different, but the last lines should be the same:

```
$ tail -n 4 locale/fr/LC_MESSAGES/tutorial.po
#: templates/layout.pt:10
msgid "Contact me!"
msgstr ""
```

You may edit this new `tutorial.po` file and add a translation (see the highlighted line below):

```
# French translations for UNKNOWN.
# Copyright (C) 2012 ORGANIZATION
# This file is distributed under the same license as the UNKNOWN project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2012.
#
msgid ""
msgstr ""
"Project-Id-Version: UNKNOWN 0.0.0\n"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"
"POT-Creation-Date: 2012-06-17 15:05+0200\n"
"PO-Revision-Date: 2012-06-17 15:09+0200\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: fr <LL@li.org>\n"
"Plural-Forms: nplurals=2; plural=(n > 1)\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 0.9.6\n"

#: templates/layout.pt:10
msgid "Contact me!"
msgstr "Contactez-moi !"
```

Once the `tutorial.po` file is ready, you have to compile it (into a `tutorial.mo` file) with the `compile_catalog` command:

```
$ python setup.py compile_catalog
running compile_catalog
1 of 1 messages (100%) translated in 'locale/fr/LC_MESSAGES/tutorial.po'
compiling catalog 'locale/fr/LC_MESSAGES/tutorial.po' to 'locale/fr/LC_MESSAGES/
↪tutorial.mo'
```

If you change your template and add new messages or change existing messages, you will need to perform a similar set of commands:

```
$ # change template and add or modify messages to translate
$ python setup.py extract_messages
$ python setup.py update_catalog
$ # edit your updated '.po' file
$ python setup.py compile_catalog
```

Note that you need to run the `init_catalog` command only once. Afterwards, you will have to run the `update_catalog` command.

Note: As indicated above, you do not have to use Babel and Lingua to generate the translation files. You may very well create the `.po` by hand and generate the compiled `.mo` file with `msgfmt`. However, having tools (be it Babel and Lingua or others) extract messages from your templates and metadata files is very valuable and can save a lot of time.

This is all good, we have translated our message. But we still need to indicate to Soho that the `en` section has content in English and the `fr` section has content in French, otherwise the template will not know which language to translate the "Contact me!" message to. The language should be set in a metadata file, under the `locale` binding. Since all files of the `en` directory are indeed in English, we can set the metadata on the directory itself (as a file named `en/.meta.py`) and it will be inherited by all its files:

```
locale = 'en'
```

As well, we create a similar file in `fr/.meta.py`:

```
locale = 'fr'
```

The template still expects a `title` key in the metadata, so we need to indicate this for each source file. For example, in `en/index.html.meta.py`:

```
title = 'My home page'
```

Let's check that we have all expected files:

```
$ ls -R
assets    locale    setup.cfg    setup.py    sohoconf.py    src    templates    www

./assets:
css

./assets/css:
style.css

./locale:
fr    tutorial.pot

./locale/fr:
LC_MESSAGES

./locale/fr/LC_MESSAGES:
tutorial.mo    tutorial.po

./src:
en    fr

./src/en:
contact.html    contact.html.meta.py    index.html    index.html.meta.py
```

```
./src/fr:
contact.html    contact.html.meta.py    index.html    index.html.meta.py

./templates:
layout.pt
```

All right, we can generate our web site:

```
$ soho-build
<date> - INFO - Copying assets...
<date> - INFO - Copying "/path/to/site2/assets/css/style.css" to "/path/to/site2/www/
↪css/style.css"
<date> - INFO - Building HTML files...
<date> - INFO - Processing "/path/to/site2/src/en/contact.html" (writing in "/path/to/
↪site2/www/en/contact.html").
<date> - INFO - Processing "/path/to/site2/src/en/index.html" (writing in "/path/to/
↪site2/www/en/index.html").
<date> - INFO - Processing "/path/to/site2/src/fr/contact.html" (writing in "/path/to/
↪site2/www/fr/contact.html").
<date> - INFO - Processing "/path/to/site2/src/fr/index.html" (writing in "/path/to/
↪site2/www/fr/index.html").
<date> - INFO - Generating Sitemap...
<date> - INFO - Done.
```

We can see that the “Contact me” message appears in the proper language in each section of the site:

```
$ head -n 12 www/en/index.html | tail -n 3
<div class="footer">
 <a href="contact.html">Contact me!</a>
</div>
$ head -n 12 www/fr/index.html | tail -n 3
<div class="footer">
  <a href="contact.html">Contactez-moi !</a>
</div>
```

Note: If you open the HTML pages in your browser (as in `file:///path/to/site2/www/en/index.html`), you may find that the CSS is not loaded. This is because we indicated an absolute path (`/css/style.css`) instead of a relative path to cope with the multi-level structure of the site. You would need an HTTP server to have the CSS correctly loaded. Fortunately, Python comes with a very handy simple HTTP server that you can run with the following command:

```
$ cd $SITE_ROOT
$ cd www
$ python -m SimpleHTTPServer 8000
```

You can then see your site at http://localhost:8000/en/index.html.

Reference

This chapter provides an exhaustive list of all configuration settings, command-line options and template bindings of Soho. If you did not do so yet, you may want to have a look at the *Tutorial* first.

Configuration file settings

The configuration file (usually named `sohoconf.py`) is a standard Python module that may define the following variables:

`asset_dir` The directory where assets (images, stylesheets, etc.) live. Must be set to `None` if no such directory exists.

Default: `'./assets'`.

`assets_only` If set, process only assets, not source files. This may be useful if the only changes are on the CSS, for example.

Default: `False`.

`base_url` The base URL of the web site. This is used only to generate the URLs in the Sitemap. If you want the Sitemap to have valid URLs, this variable must be set.

Default: `'http://exemple.com/soho/default-base-url'`

`do_nothing` If set, no directories nor files are created. This can be useful to test a new configuration. Note that you can combine this setting with `force` (see below): nothing will be created either.

Default: `False`

`force` If set, force the generation of HTML files, even if they have already been generated and are up to date. Note that you can combine this setting with `do_nothing` (see above).

Default: `False`

`hide_index_html` If set, `/index.html` suffixes are removed from:

- the `path` key that is automatically added in the `md` binding passed to the template (see *Template bindings (metadata)* below);
- URLs in the Sitemap (if a Sitemap is generated).

Default: `True`

`ignore_files` A (possibly empty) sequence of regular expressions. If the path of a file matches one of these expressions, it will not be processed.

Default: `('.*\.DS_Store$', '.*~$')`

`locale_dir` The directory where translations are stored. Must be set to `None` if no such directory exists.

Default: `'./locale'`.

`logger_level` The minimum level of the messages that will be logged. Must be one of `debug`, `info`, `warning` or `error`.

Default: `'info'`.

`logger_path` The path to the log file. If set to `-` (a single dash), messages will be logged on the standard output. If a path is provided, it must be an absolute path.

Default: `'-'`

`out_dir` The directory where the web site will be generated. This directory will be created if it does not exist.

Default: `'./www'`

`src_dir` The directory where source files live.

Default: `'./src'`

sitemap The name of the Sitemap file. Must be set to `None` if you do not want such a file to be generated.

Default: `'sitemap.xml'`

template The filename of the template to use. It must not be a relative or absolute path to the file (like `/path/to/templates/layout.pt`) but only a filename (`layout.pt`).

template_dir The directory where templates live.

Default: `'./templates'`

Note: All directories (except `www_dir`) are expected to exist. If you do not need the feature, you must set the path to `None`.

Also, each directory may be an absolute path or a relative path. If it is a relative path and starts with `./` (more precisely a dot followed by the separator on your operating system), the path is supposed to be relative to the directory of the configuration file. For example, `'./www'` will indicate a directory named `www` next to the configuration file. If the given path is a relative path and does not start with `./`, then the path will be relative to the working directory.

Default values can be found in the `soho.defaults` module and reused in your own configuration file:

```python
import soho.defaults

ignore_files = soho.defaults.DEFAULT_IGNORE_FILES + (
    # Sass source files and cache
    'assets/css/src',
    'assets/css/.sass-cache')
```

Command-line options

`soho-build` accepts the following command-line options:

-a, --assets-only See `assets_only` setting above.

-c CONFIG_FILE Use `CONFIG_FILE` as the configuration file. By default, Soho uses a file named `sohoconf.py` in the working directory.

-d, --dry-run, -do-nothing See `do_nothing` setting above.

-f, --force See `force` setting above.

-h, --help Show all command-line options.

-v, --version Show the version number.

Template bindings (metadata)

For each source file that is processed, the template receives two bindings:

body The HTML fragment as a string.

md A dictionary that contains the specific metadata of the source file (which inherits from the metadata of its directory, recursively) plus the following key:

path The URL to the generated file relative to the root of the site. It always starts with a `/`.

For example, the source file in `src/foo/bar/file.html` would have a path equal to `/foo/bar/file.html`.

Development

Soho is hosted on GitHub at https://github.com/dbaty/Soho. Feel free to report bugs and provide feedback there. Soho has a test suite that contains both unit tests, integration tests and functional tests which you may run with `make test` (that uses your own Python) or `tox` (that uses Python 2.7 and Python 3.2).

Soho is written by Damien Baty and is licensed under the 3-clause BSD license, a copy of which is included in the source and reproduced below:

List of releases of Soho

unreleased

- Fix bug in tests that caused functional tests to fail. This was due to the fact that "git clone" does not preserve file modification time: this caused test failures because we were diff'ing "sitemap.xml" files that do contain last modification time.

Soho 0.8.0 (2012-08-17)

Major rewrite. Old Soho projects should be easily moved to the new format (see documentation).

Soho 0.7 (2008-02-17)

First public release. Note that it was registered in PyPI as "soho" (with a lowercase "s").

API Documentation

`soho.builder` module

`soho.generators` package

class `soho.generators.`**`BaseGenerator`**

The base class that any generator must implement.

`generate`(*path*)

Return a tuple that consists of the metadata and the HTML fragment generated from the file at the given `path`.

`soho.generators.`**`register_generator`**(*spec*, **ext*)

Register a generator.

`spec` a string that represents the full path to a class, for example `'soho.generators.rst.RSTGenerator'`. The class must implement the same interface as *`soho.generators.BaseGenerator`*.

`ext` one or more file extensions to which the plugin will be associated. At least one file extension must be provided. File extensions should not contain the dot, for example `'html'`, not `'.html'`.

`soho.renderers` package

class `soho.renderers.`**`BaseRenderer`**(*template_path*)

The base class that any renderer must implement.

There is only one renderer for now, so the API is subject to change (as soon as a second renderer is implemented).

`render`(***bindings*)

Render the template with the given `bindings`.

`soho.renderers.`**`register_renderer`**(*spec*, **ext*)

Register a renderer.

`spec` a string that represents the full path to a class, for example `'soho.renderers.zpt.ZPTRenderer'`. The class must implement the same interface as *`soho.renderers.BaseRenderer`*.

`ext` one or more file extensions to which the plugin will be associated. At least one file extension must be provided. File extensions should not contain the dot, for example `'html'`, not `'.html'`.

Python Module Index

Index