blot Documentation

Release 0.1.0

Dustin Lacewell

August 01, 2016

Blot is a static site generator built upon an idea of generalized content processing. It has no pre-concieved notions of the kinds of content sources you have or how they should be utilized in site generation. Instead, the problem of site-generation is modeled as a general problem of content transformation. In Blot the problem model is something like:

- What types of content are there?
- For a given content type, where should we get the content?
- How should content sources be parsed into assets?
- What kinds of asset transformations should be made?
- How do assets relate to output content?
- How should output content be rendered?
- How should output content be written?

At a high-level this problem can be broken into two steps:

- **Content Reading** where content sources are discovered, parsed and processed. The result is a context object containing all the resulting content assets.
- **Asset Writing** where the resulting file-system location of assets are determined and they are rendered to disk.

Blot site-configuration reflects this process and consists of defining pipelines that answer all of these questions. Luckily, nice abstractions make it easy for you to define such pipelines for your needs.

CHAPTER 1

Installation

To install Blot you can use pip:

```
pip install blot
```

Or you can clone it from github:

```
git clone https://github.com/dustinlacewell/blot.git
cd blot && python setup.py install
```

CHAPTER 2

Basic Concepts

With a typical site generator you would probably write a declarative configuration file that specified to the generator framework a bunch of settings like where your posts are, how you want categories and tags setup and so on. It may even be flexible enough to let you specify where your content should be loaded from and how it should be layed out when rendered.

However, you're usually specifying essentially configuration variables to built-in mechanisms within the framework. If there is no configuration expressable to get those mechanisms to generate the site precisely how you'd like the only usual solutions are plugin systems or forking.

2.1 Composable Pipelining

What if we could ease out those internal mechanisms and make them composable? If we could, we wouldn't be limited only to the behaviors available from built-in mechanisms. We can simply use a different mechanism that serves the same role but more how we'd like.

This results in a much more procedural build process. Instead of writing a long list of configuration variables, you're specifying the mechanisms of the pipeline for each kind of content you have.

2.2 Site Generation Process

Site generation is modeled as a generic process of "content transformation". A site may contain various kinds of content:

- chronological articles
- static pages
- structured data
- media assets (css, js, images)

For the purposes of site generation, the concerns for each of these content types are the same:

- where is the content source coming from?
- how is the content source parsed?
- how should parsed content assets be processed?
- how should output content be rendered?
- where should output content be written?

In each case the need to load some stuff from disk, maybe do stuff to it, maybe render it into something else, and to write it somewhere is universal. We can visualize the pipeline for a given content type as:

```
+-------------------+        +------------------+       +------------------+
|                   |        |                  |       |                  |
|  Content Sources  +------> |  Content Assets  +-----> |  Output Content  |
|                   |        |                  |       |                  |
+-------------------+        +------------------+       +------------------+
```

2.3 Asset Processing

Once content sources have been loaded from disk and potentially parsed, the resulting content assets are objects that contain the content and optional metadata. Its easy to imagine then, those assets finding their way to the disk at the end of the pipeline.

For example, we might have some articles written in Markdown, which are loaded and parsed into content assets. Those articles are then written out as html in a convienent place. But what about other kinds of content that don't have source files on disk? Categories are good example.

If you have a category assigned in the metadata of each article, you probably want pages generated that index the articles belonging to that category. But so far, we've only seen that content assets that come from content sources can be written out. However, during asset processing, additional asset types can be generated. This allows your content sources to "fan out" to multiple logical asset types, each with its own distinct output configuration:

```
                             +------------------+       +------------------+
                             |                  |       |                  |
                             | Generated Assets +-----> |  Output Content  |
                             |                  |       |                  |
                             +---------^--------+       +------------------+
                                       |
+-------------------+        +------------------+       +------------------+
|                   |        |                  |       |                  |
|  Content Sources  +------> |  Content Assets  +-----> |  Output Content  |
|                   |        |                  |       |                  |
+-------------------+        +------------------+       +------------------+
                                       |
                             +---------v--------+       +------------------+
                             |                  |       |                  |
                             | Generated Assets +-----> |  Output Content  |
                             |                  |       |                  |
                             +------------------+       +------------------+
```

2.4 Output Configurations

Similarly to the "fan out" possible between content sources and the asset types generated from them; multiple output configurations can target the same asset types. It should start to be apparent the flexibility of Blog's ability to generate static content:

```
                                                        +------------------+
                                                        |                  |
                                                    +---> Category Index   |
                             +------------------+   |   |                  |
                             |                  |   |   +------------------+
                             | Category Assets  +---+
                             |                  |   |   +------------------+
                             +---------^--------+   |   |                  |
                                       |            +---> Category Details |
                                       |                |                  |
                                       |                +------------------+
                                       |
                                       |                +------------------+
                                       |                |                  |
                                       |            +---> Article Index    |
+-------------------+        +------------------+   |   |                  |
|                   |        |                  |   |   +------------------+
| Markdown Sources  +------> |  Article Assets  +---+
|                   |        |                  |   |   +------------------+
+-------------------+        +------------------+   |   |                  |
                                       |            +---> Article Details  |
                                       |                |                  |
                                       |                +------------------+
                                       |
                                       |                +------------------+
                                       |                |                  |
                                       |            +---> Tag Index        |
                             +---------v--------+   |   |                  |
                             |                  |   |   +------------------+
                             |    Tag Assets    +---+
                             |                  |   |   +------------------+
                             +------------------+   |   |                  |
                                                    +---> Tag Details      |
                                                        |                  |
                                                        +------------------+
```

CHAPTER 3

Reading Content

The first phase of the build process involves the loading and processing of source content and processed assets. The goal of this phase is to produce the context dictionary that will be used in the second phrase of writing.

3.1 Base Context

The reading phase starts with the creation of a base context dictionary. This dictionary contains static information that may be useful to inside the templates of template oriented writers. Another possibility is of pipeline components basing their behavior on top-level values in the context. This is generally avoided when possible, favoring direct configuration of pipeline components, but some components do use top-level context variables as a convience in configuring many of them in a single place.

No keys in of the base context that share a name with any of your content types should be used, since this is where the assets of those content types will be stored.

3.2 Content Types

Before any part of the reading phase can begin a definition of your content types must be provided. This definition is provided as a dictionary of dictionaries. Each top-level key maps the name of a content type to its definition. Each type definition has three required keys:

- `loader` discovers the paths of candidate content sources
- `reader` parses the content of discovered sources into assets
- `processors` a list of asset processors to apply in order

A simple example for loading some Markdown articles might look something like:

```
content_types = {
  'articles': {
    'loader': blot.loaders.BasicLoader('content/articles', ['md]),
    'reeader': blot.readers.MarkdownReader(),
    'processors': [
      blot.assets.PathMetadata(),
      blot.assets.AutoSummary(),
      blot.assets.Humanizer('date'),
    ]
  }
}
```

3.3 Producing the Write Context

Once a base context and content types have been defined the reading process can be performed with `blot.read()`:

```
write_context = blot.read(base_context, content_types)
```

3.3.1 Loaders

Loaders are responsible for discovering source content on disk and determining which of those sources are relevant to its content type.

BasicLoader

Blot comes with a fairly simple loader that finds files on disk and filters them based on extension and some simple exclusion and inclusion rules.

class `blot.loaders.`**`BasicLoader`**(*path*, *excludes=[]*, *includes=[]*, *extensions=[]*)

Includes basic loading discovery functionality featuring exlcude and include regular-expressions.

Takes a path and recursively locates all of the files within it. Any paths that match any exclusion expressions are not returned. Any paths that match any inclusion expression are included regardless of whether they match an exclusuion.

3.3.2 Readers

Readers are responsible for parsing the content sources passed on by the loader. For each content source, the Reader attempts to produce a `blot.assets.ContentAsset`:

class `blot.assets.`**`ContentAsset`**(*source=None*, *content=None*, *metadata={}*)

An object with some data and associated metadata.

ContentAssets represent the result of loading and reading content sources. The result of parsing the source source provides its content and optional metadata dictionary.

ContentAssets are fed through a pipeline of processors where they may be modified in various ways. Eventually, they are passed to Writers which can render them to their final destinations on disk.

A basic dictionary-like access is avaiable as a shortcut to the asset's metadata. Once the assets's *target* attribute has been set, its *url* property will become available.

Some assets are produced by the processing of other assets and have no source.

StaticReader

The simplest reader in Blot is the `blot.readers.staticreader.StaticReader` which performs no parsing on source content. Therefore it also produces no metadata.

class `blot.readers.`**`StaticReader`**(*asset_class=<class 'blot.assets.base.ContentAsset'>*)

Simple reader which performs no content or metadata parsing.

MarkdownReader

A reader which parses source content as Markdown. If the content source contains a "fenced block" of key-value properties, these will be parsed as the metadata for the asset:

```
---
title: Awesome Post!
---

Reasons to read this post:
  - it is awesome!
  - because!
```

class `blot.readers.`**`MarkdownReader`**(*asset_class=<class 'blot.assets.base.ContentAsset'>, extras=['metadata', 'fenced-code-blocks']*)

Reader which parses a content source as Markdown.

Metadata should be specified by a "fenced" yaml block at the top of the file. Metadata keys will be lower-cased. Duplicate keys will override ones earlier in the file.

3.3.3 Processors

After all the content sources have been parsed by readers into content assets they are fed through a pipeline of processors. Each processor defined for the content type are in order.

Processing Context

For each content type a private context dictionary is created while processing for that content type takes place. The content assets produced by the reader are stored under the `assets` key. As each processor is ran, it receives this context dictionary and can retrieve the content type's assets through that key.

Asset Modification

One use of processors is to iterate over the content assets and modify them in some way. The changes made by one processor will be visible processors later in the pipeline. In this way, changes introduced by asset processing are cummulative.

Context Modification

Processors are also free to change the content type context dictionary in anyway, including adding or removing keys or modifying the values that they point to.

Generated Assets

Processors may also produce completely new sets of content assets and store them under keys in the context dictionary. These assets can be targetted for output just like the core content type assets.

3.3.4 Basic Processors

AutoSummary

blot.assets.AutoSummary will attempt to parse asset contents as HTML and automatically generate a summary. The summary will be stored on the asset's `summary` metadata property by default.

Humanizer

blot.assets.Humanizer will apply a humanizing transformation to date, time and other numerical metadata properties.

PathMetadata

blot.assets.PathMetadata will derive a number of metadata properties from various aspects of the asset's source path.

Slugifier

blot.assets.Slugifier will slugify a named metadata property and store the result on another metadata property.

3.3.5 Aggregation Processors

A certain kind of processor is useful for generating new types of "aggregate assets" from those being processed. These new assets contain a list of all the original assets that share a specific metadata property value. These generated *blot.assets.base.Aggregate* based assets can then be targetted for output independently from the original assets.

A concrete use-case would be a site that featured articles from multiple authors. It might be useful to generate a list of articles written by each author. A `blot.assets.Aggregator` based processor could do just that. Configured to aggregate an `author` metadata propety a new set of aggregate assets would be generated for each unique author. It might be then desirable to an index page of featured authors and detail pages for each author listing the articles they've written. This is no problem because generated assets can be targeted for writing the same as core content type assets.

Categories

blot.assets.Categories will aggregate assets across their `category` metadata property value by default. Generated category aggregates will be stored in the content type context dictionary under the `categories` key. Each aggregated asset in a category will have its `category` property updated with the corresponding Category instance.

Series

blot.assets.Series will aggregate assets across their `series` metadata property value by default. Generated series aggregates will be stored in the content type context dictionary under the `series` key. Each aggregated asset in a series will have its `series` property updated with the corresponding Series instance. Unlike Categories, Series are only generated if it contains 2 or more aggregated assets.

Tags

blot.assets.Tags will aggregate assets across their `tags` metadata property value by default. The metadata property is split on a comma and aggregation takes place for each individual tag. Generated tag aggregates will be stored in the content type context dictionary under the `tags` key. Each aggregated asset that has at least one tag will have its `tags` property updated with a list of corresponding Tag instances.

CategoryTags

blot.assets.CategoryTags will attempt to aggregate all of the Tags for all assets in a given Category and store this superset of tags on the Category's `tags` metadata property.

SeriesTags

blot.assets.SeriesTags will attempt to aggregate all of the Tags for all assets in a given Series and store this superset of tags on the Series' `tags` metadata property.

CHAPTER 4

Writing Assets

Once all content types have had their assets processed a final context dictionary will be produced. A key in the dictionary corresponding to each content type maps to the content type specific context dictionary that was produced during the processing of each content type. In side each content type context, the `assets` key contains the content type's assets. Additionaly, the processors assigned to the context types may have also populated various keys with generated asset types:

```
render_context = {
  'articles': {
    'assets': [...],
    'categories': [...],
    'tags': [...],
  },
  'staticfiles': {
    'assets': [...],
  },
  GLOBAL_VARIABLE_1: "some value",
  GLOBAL_BOOLEAN: False,
}
```

Writing involves assigning writers to the various asset types, whether they be core content types or generated ones, to get them written to the filesystem where desired. The writers do this in two steps:

- **Targeting**: Figure out where some assets should be rendered
- **Rendering**: Rendering assets to their targetted locations on disk

4.1 Targeting

Each content asset as a non-metadata attribute `target` which is for storing the location on disk where the asset should be written to disk. Since Blot a static *site* generator, this location determines its URL as well. During the writing phase, all writers perform targeting on all of their assets before any writer performs rendering. This is to ensure that during rendering, the URLs of all asset types are available. Otherwise, only the URLs of assets already processed could be properly rendered.

If a writer will be rendering output for each targeted asset the target path pattern will generally contain one or more interpolation variables that name metadata properties. This allows the output path for each asset to vary uniquely so they don't overwrite each other.

4.2 Rendering

Now that all assets have been properly targeted each writer undergoes rendering wherein it may process the content of assets for the last time. This will typicall involve combining the asset, with the global context and some specified template to produced a rendering of the asset. However, this isn't nessecary writers are free to do whatever or nothing at all. At this point the writer shouldn't be modifying assets or the global context at all.

Once the final content of an asset has been derived it is ready for writing to its target destination.

4.3 Performing the Writes

With the rendering context in hand *blot.write()* can be used to actually generate your site:

```
posts = context['posts']['assets']
categories = context['posts']['categories']
staticfiles = context['staticfiles']['assets']

writers = [
  blot.writers.ListWriter(posts, 'post.html', 'posts/{slug}/index.html'),
  blot.writers.IndexWriter(posts, 'posts.html', 'posts/index.html'),
  blot.writers.ListWriter(categories, 'category.html', '/{name}.html'),
  blot.writers.IndexWriter(categories, 'categories.html', 'index.html'),
  blot.writers.StaticWriter(staticfiles, 'static/{filename}'),
]

blog.write(context, writers)
```

4.3.1 Writers

Writers are responsible for performming asset targetting and rendering. Some writers produce multiple output files some produce a single ouputs. Some writers involve templated rendering that involve the global context but that isn't nessecary.

The same assets may be passed to multiple writers, but only one targeting writer per set of assets should be used. If multiple targeting writers are used, only the targeting that is performed last will apply.

4.3.2 Targetting Writers

Targeting writers perform targeting on the assets passed to them, updating their *target* properties. This affects how links to assets are generated.

ListWriter

`blot.writers.ListWriter` will perform targeting on each asset based on the provided path pattern. For each asset, its metadata will be used to interpolate the path pattern and this should result in a path unique to the asset. If not, assets will overwrite each other. The output will be the result of rendering the specified template against the global context. The following context updates are made for each asset's rendering:

`assets` a list of all the assets the type shared by the currently rendering asset

`asset` the currently rendered asset

4.3.3 Non-targetting Writers

Since Non-targetting writers do not update asset targets it is safe to use multiple of them against any given assets.

StaticWriter

`blot.writers.StaticWriter` writes each asset without any rendering of the content whatsoever. And so it takes only a target path pattern that should contain some differentiating interpolation variables.

IndexWriter

`blot.writers.IndexWriter` should take a non-interpolated destination path where the provided template will be rendered. This is the only output file and it is rendered with only a single context update:

> `assets` a list of all the writer's assets

Additionally, during targeting this writer will update the global context with a key specified by *variable_name* that points to the URL linking to the output file making it available globally to rendering of all writers.

PaginatedWriter

`blot.writers.PaginatedWriter` takes an interpolatable destination path pattern. This pattern requires the inclusion of an interpolation variable named *{page}*.

Based on the *size* parameter, this writer will render multiple outputs based on simple pagination. For each page it will render the given template with the following context updates:

> `assets` the subset of assets in the current page `page_number` the current page number `first_page` url to the first page `last_page` url to the last page `previous_page` url to the page before the current one `next_page` url to the page after the current one

blot

`blot.read`(*context*, *content_types*)
Stage 1 of site generation.

For each defined content type:

- use loader to discover input content
- use reader to parse input content into content assets
- run each processor against the assets
- return resulting context dictionary containing everything

`blot.write`(*context*, *writers*, *build_path='output'*)
Stage 2 of site generation.

For each specified writer:

- Resolve each asset's target output location on disk

For each specified writer:

For each asset fed to the writer:

- Render the asset using the global context
- Write the asset to disk

The reason for targetting all assets before rendering is so that during rendering the URLS for all assets are available. This allows any templates being used to render a specific content type to include links to other assets of other content types.

5.1 Subpackages

5.1.1 blot.assets

Submodules

blot.assets.autosummary

class `blot.assets.autosummary.`**`AutoSummary`**(*key='summary'*)
Bases: `object`

Generates a summary from an asset's content.

process (*context*)

blot.assets.base

class `blot.assets.base.`**Aggregate** (*name*, *assets=None*)

Bases: *blot.assets.base.ContentAsset*

A special type of ContentAsset that has two known metadata properties:

- name : The metadata value of some other ContentAsset this Aggregate is based on
- assets : The list of ContentAssets in which this Aggregate appears

class `blot.assets.base.`**Aggregator** (*key*, *pattern*, *asset_class=<class 'blot.assets.base.Aggregate'>*)

Bases: `object`

Base-class for an asset processor that generates Aggregates for each value in the metadata of processed ContentAssets.

When used as an asset processor, it will look at a specific metdata property of each ContentAsset. For each unique value found at this property, a new Aggregate asset will be generated.

Each Aggregate object contains in its own metadata an *assets* key that contains a list of each of the ContentAssets in which the aggregated value was found.

It is up to subclasses of Aggregator to make the list of generated Aggregate assets available. Subclasses also have the chance to update each aggregate and the asset it comes from.

Generally, the original content type's context will be updated with a list of the Aggregates under a relevant key in the *finish* method. Check the Tags and Categories processors for examples.

finish (*context*, *aggregates*)

get_values (*asset*)

Get the aggregated property value from the asset.

process (*context*)

Check each asset in the content Type context and aggregate discovered values over the specific metadata key into new assets.

process_aggregate (*aggregate*, *asset*)

class `blot.assets.base.`**ContentAsset** (*source=None*, *content=None*, *metadata={}*)

Bases: `object`

An object with some data and associated metadata.

ContentAssets represent the result of loading and reading content sources. The result of parsing the source source provides its content and optional metadata dictionary.

ContentAssets are fed through a pipeline of processors where they may be modified in various ways. Eventually, they are passed to Writers which can render them to their final destinations on disk.

A basic dictionary-like access is avaiable as a shortcut to the asset's metadata. Once the assets's *target* attribute has been set, its *url* property will become available.

Some assets are produced by the processing of other assets and have no source.

get (*key*, *default=None*)

url

blot.assets.categories

class `blot.assets.categories.`**`Categories`**(*key='category'*, *pattern='(.*)'*)

Bases: *`blot.assets.base.Aggregator`*

A simple single value Aggregator for organizing content assets.

`finish`(*context*, *categories*)

`process_aggregate`(*category*, *asset*)

blot.assets.categorytags

class `blot.assets.categorytags.`**`CategoryTags`**

Bases: `object`

Processor that adds to each Category for a content type, all the Tags of its assets.

`process`(*context*)

blot.assets.humanizer

class `blot.assets.humanizer.`**`Humanizer`**(*attr='date'*)

Bases: `object`

Processor which runs *naturaltime* over the specified metadata property.

`process`(*context*)

blot.assets.metadata

class `blot.assets.metadata.`**`PathMetadata`**

Bases: `object`

Generates a number of metadata properties based on an asset's source path.

`process`(*context*)

blot.assets.series

class `blot.assets.series.`**`Series`**(*key='parent'*, *pattern='(.*)'*, *part_key='filename'*, *part_pattern='(.*)'*)

Bases: *`blot.assets.base.Aggregator`*

Aggregator which attempts to automatically discover multi-part content series.

Like other Aggregators the Series processor generates a new set of assets based on values extracted from a specified key of some target assets. However, the Series processor will only generate new assets for aggregate values found across two or more target assets.

For each Series generated a number of metadata properties are added to target assets. These properties contain the first, last, previous and next target assets in the Series. This allows a theme to construct navigation between each asset in the Series.

In addition to the normal aggregation metadata property, the Series processor requires direction for deriving the order of assets in the Series. So the *part_key* and *part_pattern* are used to select this value from the metadata of target assets. If the extracted part value is 'index.html' it will automatically be converted to 1, putting it as the first part in the series.

finish (*context*, *series*)

get_part_number (*asset*)
derive the part number of an asset

process_aggregate (*series*, *asset*)
generate a title for the series itself

blot.assets.seriestags

class `blot.assets.seriestags.`**SeriesTags**
Bases: `object`

Processor that adds to each Series for a Content Type, all the Tags of its assets.

process (*context*)

blot.assets.slugifier

class `blot.assets.slugifier.`**Slugifier** (*source_attr='title'*)
Bases: `object`

Processor that slugifies an asset metadata property.

process (*context*)

blot.assets.tags

class `blot.assets.tags.`**Tags** (*key='tags'*, *pattern='([^,]+)'*)
Bases: *blot.assets.base.Aggregator*

Simple multi-value Aggregator which splits the named metadata property by comma.

finish (*context*, *tags*)

process_aggregate (*tag*, *asset*)

blot.assets.titler

class `blot.assets.titler.`**Titler** (*source_attr*)
Bases: `object`

Processor that titlecases an asset metadata property.

process (*context*)

5.1.2 blot.loaders

Submodules

blot.loaders.basicloader

class `blot.loaders.basicloader.`**`BasicLoader`**(*path*, *excludes=[]*, *includes=[]*, *extensions=[]*)

Bases: `object`

Includes basic loading discovery functionality featuring exlcude and include regular-expressions.

Takes a path and recursively locates all of the files within it. Any paths that match any exclusion expressions are not returned. Any paths that match any inclusion expression are included regardless of whether they match an exclusuion.

`check_path`(*path*)

`find_files`()

`load`()

Return a list of all paths discovered and allowed by the exclusion and inclusion expresions.

5.1.3 blot.readers

Submodules

blot.readers.markdownreader

class `blot.readers.markdownreader.`**`MarkdownReader`**(*asset_class=<class 'blot.assets.base.ContentAsset'>*, *extras=['metadata', 'fenced-code-blocks']*)

Bases: `object`

Reader which parses a content source as Markdown.

Metadata should be specified by a "fenced" yaml block at the top of the file. Metadata keys will be lower-cased. Duplicate keys will override ones earlier in the file.

`read`(*paths*)

`read_path`(*path*)

blot.readers.staticreader

class `blot.readers.staticreader.`**`StaticReader`**(*asset_class=<class 'blot.assets.base.ContentAsset'>*)

Bases: `object`

Simple reader which performs no content or metadata parsing.

`read`(*paths*)

5.1.4 blot.writers

Submodules

blot.writers.indexwriter

class `blot.writers.indexwriter.`**`IndexWriter`**(*assets*, *variable_name*, *template*, *path*)

Bases: `object`

Writer that renders a single jinja2 template. The global context is updated with a URL to the rendered file during targetting. All assets are passed into the context as *assets* during rendering.

`render`(*context*)

`target`(*context*)

blot.writers.listwriter

class `blot.writers.listwriter.`**`ListWriter`**(*assets*, *template*, *path*)

Bases: `object`

Writer that renders a jinja2 template for each targetted asset.

`render`(*context*)

`target`(*context*)

blot.writers.paginatedwriter

class `blot.writers.paginatedwriter.`**`PageHelper`**(*items*, *size*)

Bases: `object`

Utility class that providers an iterator that returns slices of the underlying list.

class `blot.writers.paginatedwriter.`**`PaginatedWriter`**(*assets*, *variable_name*, *template*, *path*, *size=10*)

Bases: `object`

Writer that paginates the underlying assets and renders a template for each page.

PaginatedWriter takes divides its assets into groups of the specified size and renders the specified template for each group. The global context variable will point to the URL of the first page.

The context when rendering each of the pages contains a number of variables for navigating between the pages.

`pathfor`(*index*)

generate the target path for a given page

`render`(*original_context*)

`target`(*context*)

blot.writers.staticwriter

class `blot.writers.staticwriter.`**`StaticWriter`**(*assets*, *path*)

Bases: `object`

Writer that writes asset content without rendering.

render (*context*)

target (*context*)

5.2 Submodules

5.2.1 blot.summarize

class `blot.summarize.`**Summary** (*url*, *article_html*, *title*, *summaries*)
Bases: `object`

`blot.summarize.`**compare_sents** (*sent1*, *sent2*)
Compare two word-tokenized sentences for shared words

`blot.summarize.`**compare_sents_bounded** (*sent1*, *sent2*)
If the result of compare_sents is not between LOWER_BOUND and UPPER_BOUND, it returns 0 instead, so outliers don't mess with the sum

`blot.summarize.`**compute_score** (*sent*, *sents*)
Computes the average score of sent vs the other sentences (the result of sent vs itself isn't counted because it's 1, and that's above UPPER_BOUND)

`blot.summarize.`**find_likely_body** (*b*)
Find the tag with the most directly-descended <p> tags

`blot.summarize.`**is_unimportant** (*word*)
Decides if a word is ok to toss out for the sentence comparisons

`blot.summarize.`**only_important** (*sent*)
Just a little wrapper to filter on is_unimportant

`blot.summarize.`**summarize_block** (*block*)
Return the sentence that best summarizes block

`blot.summarize.`**summarize_blocks** (*blocks*)

`blot.summarize.`**summarize_page** (*url*)

`blot.summarize.`**summarize_text** (*text*, *block_sep='\n\n'*, *url=None*, *title=None*)

`blot.summarize.`**u** (*s*)
Ensure our string is unicode independent of Python version, since Python 3 versions < 3.3 do not support the u"..." prefix

5.2.2 blot.utils

`blot.utils.`**generate_cloud** (*items*, *minsize=0.8*, *maxsize=1.0*)
From a list of Aggregates return a list of tuples containing each Aggregate and its relative "size" in a tag cloud normalized to 1.0.

This is useful for rendering into a <span> tag's *style* attribute to affect font-size.

`blot.utils.`**get_values** (*asset*, *key*, *pattern='(*.)'*)
Returns values extracted from a metadata property of an asset named by key.

`blot.utils.`**pathurl** (*path*)
Returns a nicer URL for the path on disk for webservers that treat URLs missing a filename to be requesting index.html

If the path ends in */index.html*, make it end with */*.

Make every path absolute.

`blot.utils.`**`render`**(*path*, *template*, *context*)

Uses Jinja2 to render a template relative to the path with the given context.

CHAPTER 6

Building a Blog

This tutorial will showcase some of the ways to use Blot to build a personal blog. This blog will be quite simple but it should have enough features to convey how site generation works.

6.1 Getting Setup

Firstly, you'll need to install Blot.

One that's done, we recommend installing Fabric which will be useful for writing your site's build script. You can read more about Fabric at their official overview but essentially it is a Python library that helps you write scripts that perform local or remote operations. It then exposes those operations with a nifty cli tool. Brace yourself for the shortest Fabric tutorial in the world:

Step 1: Write a Python file called `fabfile.py` and put functions in it:

```
# fabfile.py
def hello(name):
    print "Hello, {}!".format(name)
```

Step 2: Use the `fab` cli tool to invoke individual functions from the fabfile:

```
$ fab -l
Available commands:

hello

$ fab hello:world
Hello, world!

Done.
```

6.2 Listing the Goals

For this tutorial we want to build just enough functionality into the blog site to showcase how Blot is used. Let's stick some typical blog features:

Chronological Posts

- Each post is a markdown file in a `posts` folder.
- There should be an index of the posts sorted chronologically

- There should be a page for each post where we can read it

Post Categories

- Each post has an assigned category
- There should be an index of the available categories
- There should be a detail page for each category listing the posts it contains

Static Assets

- There should be a logo image which we're able to integrate into our theme.
- There should be a stylesheet which styles our pages

6.3 Getting Started

Create a new directory for your site and within it create a `fabfile.py` file, a `posts` directory and a `theme` directory:

```
|-- fabfile.py
|-- posts/
`-- theme/
```

In the theme directory, create the file `style.css`. This will be our first content source. The goal with static files is simple: copy the file from the source to the destination.

Open your `fabfile.py` and enter the following contents:

```
base_context = {}
```

The `base_context` is like the starting-point for data in the build process. Anything you add here will be subject to modification by anything in the pipeline. Settings placed here can also affect the behavior of components in the pipeline. For now, we'll leave it empty.

6.4 Content Types

Add the following to `fabfile.py`:

```
content_types = {
    'stylesheets': {
        'loader': blot.loaders.BasicLoader('theme', extensions=['css']),
        'reader': blot.readers.StaticReader(),
        'processors': [
            blot.assets.PathMetadata(),
        ],
    }
}
```

The `content_types` dictionary holds all of the content type definitions for our site. Currently, there is only a single type `stylesheets`. Let's take a look at each of its keys:

6.4.1 loader

Loaders tell Blot where to find the sources of stylesheets. *blot.loaders.BasicLoader* supports some basic inclusion and exclusion rules including filtering by file extension. In this case we're loading all of the files with a `css` extension in the `theme` directory. This should target the `style.css` file.

6.4.2 reader

The reader is responsible for parsing the content of sources to produce content assets with metadata. However, *blot.readers.StaticReader* doesn't perform any parsing and returns source content as-is.

6.4.3 processors

A list of asset processors that should get a chance to modify any stylesheets returned by the loader and reader. In this case, a single asset processor `blot.assets.PathMetadata` will extract various details of the source file path and add it to the asset metadata. Things like `basename`, :code'dirname', :code'filename', :code'extension' and so on.

From this definition we have all we need in order to find stylesheets and load them as content assets.

6.5 Reading

Now that we have a base context and our content type definitions, reading can be performed with `blog.read()`. This will load our sole stylesheet, add some metadata to it. Finally, an updated version of the base context will be returned. Add the following to `fabfile.py`:

```python
def build():
    context = blot.read(base_context, content_types)
```

We now have a function in the `fabfile.py` that we can invoke from the commandline. However if we do, we wont see anything meaningful just by loading content. Let's add a quick debugging print to see the results of the reading process:

```python
def build():
    context = blot.read(base_context, content_types)
    print context
```

Now we can invoke the `build` function with the `fab` command-line utility:

```
fab build
{'stylesheets': {'assets': [<blot.assets.base.ContentAsset object at 0x7fc3d9041c50>]}}

Done.
```

We can see that the context now contains a `stylesheets` key that maps to the context for that content type. Within that context is an `assets` key that contains the actual processed content assets.

If we change the print statement to `print context['stylesheets']['assets'][0].metadata` we can get a closer look at the asset itself and evidence of *blot.assets.PathMetadata* processor at work:

```
fab build
{'ancestry': '', 'parent': 'theme', 'extension': '.css', 'basename': 'style.css', 'dirname': 'theme',

Done.
```

For more information on loaders, readers and asset processors visit the Reading Content section of the User Guide.

6.6 Writing

Writing involves taking some assets and passing them to a writer. In our case we don't need to render anything so we can use the `blot.writers.StaticWriter` to simply write our stylesheet asset contents to disk:

```
def build():
    context = blot.read(base_context, content_types)

    stylesheets = context['stylesheets']['assets']
    blot.write(context, [
        blot.writers.StaticWriter(stylesheets, 'static/{basename}'),
    ])
```

First, we grab the stylesheet assets out of the context. Then we call `blot.write()` while passing it the `context` and a list of writers. In this case, the only writer involved.

`blot.writers.StaticWriter` takes two arguments:

- The assets to write
- A path pattern describing destinations

As `StaticWriter` goes to write each asset it will determine where to write it by interpolating the asset's metadata into the provided path pattern. The path pattern we're using `static/{basename}` requires that the assets being written contain this metadata property. Luckily, `basename`, among others, is provided by the `PathMetadata` asset processor we used during reading. For our `style.css` file, we should expect its destination path to be `static/style.css`.

If we invoke the `build` function from the commandline that is exactly what we should see in the newly created `output` directory:

```
$ fab build

Done.

$ tree output
output
`-- static
    `-- style.css

1 directory, 1 file
```

For more information on writing visit the Writing Assets section of the User Guide.

6.7 Intermission

By now you should be developing a clearer idea of how you can bring together loaders, readers, processors and writers to form a pipeline for your content sources with Blot. The process for handling the rest of our content types works just like it does for our stylesheets. Really!

At this point the tutorial will start moving a lot faster.

Before moving on, let's add small helper function to `fabfile.py` to clean our output directory by deleting it. The whole script should appear as follows:

```
from fabric.api import local

import blot

base_context = {}

content_types = {
    'stylesheets': {
        'loader': blot.loaders.BasicLoader('theme', extensions=['css']),
        'reader': blot.readers.StaticReader(),
        'processors': [
            blot.assets.PathMetadata(),
        ],
    }
}


def clean():
    local("rm -r output")

def build():
    clean()
    context = blot.read(base_context, content_types)
    stylesheets = context['stylesheets']['assets']
    blot.write(context, [
        blot.writers.StaticWriter(stylesheets, 'static/{basename}'),
    ])
```

We've added a `clean` function which removes the output directory which allows us to invoke it from the commandline. It uses a function from the Fabric api, `local`, which we've imported at the top. It makes running local shell commands very convienent. As a final note, we're calling `clean` first thing from `build` which gives a clean target path each time you build.

6.8 Introducing Posts

To introduce our actual blog posts, we'll add a new `posts` content type:

```
content_types = {
    'stylesheets': {
        'loader': blot.loaders.BasicLoader('theme', extensions=['css']),
        'reader': blot.readers.StaticReader(),
        'processors': [
            blot.assets.PathMetadata(),
        ],
    },
    'posts': {
        'loader': blot.loaders.BasicLoader('posts', extensions=['md']),
        'reader': blot.readers.MarkdownReader(),
        'processors': [
            blot.assets.PathMetadata(),
        ],
    }
}
```

This is very similar to the stylesheets content type definition, but with a few changes. First, we want to load files from the `posts` directory this time and only files with a `.md` file extension. Any such files will then be parsed as markdown by the *blot.readers.MarkdownReader*.

Python Module Index