Blogofile Documentation

Release 0.7.1

Ryan McGuire

June 09, 2012

CONTENTS

Note: This documents the 0.7.1 stable release version of Blogofile. Development of an improved 0.8 version is underway at http://github.com/EnigmaCurry/blogofile, however backward compatibilty is not guaranteed. The latest version of the docs describes 0.8.

CHAPTER

ONE

INTRODUCTION

- **Definition: Blogophile (n):** A person who is fond of or obsessed with blogs or blogging.
- **Definition: Blogofile (n):** A static website compiler and blog engine, written and extended in Python.

1.1 Welcome to Blogofile

Blogofile is a static website compiler, primarily (though not exclusively) designed to be a simple blogging engine. It requires no database and no special hosting environment. You customize a set of templates with Mako, create posts in a markup language of your choice (see *Post Content*) and Blogofile renders your entire website as static HTML and Atom/RSS feeds which you can then upload to any old web server you like.

1.2 Why you should consider Blogofile

- Blogofile is **free open-source** software, released under a non-enforced MIT License.
- Blogofile sites are **fast**, the web server doesn't need to do any database lookups nor any template rendering.
- Blogofile sites are **inexpensive** to host. Any web server can host a blogofile blog.
- Blogofile is **modern**, supporting all the common blogging features:
- Categories and Tags.
- Comments, Trackbacks, and Social Networking mentions (Twitter, Reddit, FriendFeed etc), all with effective spam filtering using Disqus.
- RSS and Atom feeds, one for all your posts, as well as one per category. Easily create additional feeds for custom content.
- Syntax highlighting for source code listings.
- Ability to create or share your own plugins in your own userspace (see *Filters* and *Controllers*)
- Blogofile is **secure**, there's nothing executable on the server to exploit.
- Blogofile works **offline**, with a built-in web server you can work on your site from anywhere.
- Blogofile is **file based**, so you can edit it with your favorite text editor, not some crappy web interface.
- Seamless *Git Integration*. Publish to your blog with a simple `git push`. This also makes **backups** dirt simple.

1.3 Installing Blogofile

Blogofile is under active development, but strives to be usable and bug-free before the 1.0 release.

1.3.1 Prerequisites

Make sure you have Python and Setuptools installed. On Ubuntu you just need to run:

```
sudo apt-get install python-setuptools
```

1.3.2 Install

Download and install Blogofile with:

```
easy_install Blogofile
```

You can also get the latest development source code from github:

```
git clone git://github.com/EnigmaCurry/blogofile.git
```

CHAPTER

TWO

A QUICK TUTORIAL

Ok, if you're impatient, this is the short *short* [1] version of getting setup with blogofile.

- Install Blogofile, (see *Installing Blogofile*):

  ```
  sudo easy_install blogofile
  ```

- In a clean directory, initialize the bare bones sample site:

  ```
  blogofile init simple_blog
  ```

- Or, for a more complete sample blog (requires git):

  ```
  blogofile init blogofile.com
  ```

- Create some post files in the _posts directory. (see *Posts*)
- Build the site:

  ```
  blogofile build
  ```

- Serve the site:

  ```
  blogofile serve 8080
  ```

- Open your web browser to http://localhost:8080 to see the rendered site.

The next chapters explain this process in more detail.

[1]

- **Priest**: Do you?
- **Vespa**: Yes.
- **Priest**: Do *you*?
- **Lone Star**: I do.
- **Priest**: Good! Fine! You're married! Kiss Her!

CHAPTER

THREE

THE MAKEUP OF A BLOGOFILE SITE

Blogofile is a website compiler, but instead of translating something like C++ source code into an executable program, Blogofile takes Mako templates, and other Blogofile features, and compiles HTML for viewing in a web browser. This chapter introduces the basic building blocks of a Blogofile directory containing such source code.

3.1 An Example

The best way to understand how Blogofile works is to look at an example. Create a new directory and inside it run:

```
blogofile init simple_blog
```

This command creates a very simple blog that you can use to learn how Blogofile works as well as to provide a clean base from which you can create your own Blogofile based website.

For a more complete example, you can checkout the code for the same website you're reading right now, blogofile.com:

```
blogofile init blogofile.com
```

This command downloads the very latest blogofile.com website source code, which requires that you have git installed on your system. If you don't have it, you can just download the zip file instead.

The rest of this document will assume that you're using the simple_blog template. It is the defacto reference platform for Blogofile.

3.2 Directory Structure

Inside the source directory are the following files (abbreviated):

```
|-- _config.py
|-- _controllers
|   |-- blog
|   |   |-- archives.py
|   |   |-- categories.py
|   |   |-- chronological.py
|   |   |-- feed.py
|   |   |-- __init__.py
|   |   |-- permapage.py
|   |   `-- post.py
|-- _filters
|   |-- markdown_template.py
|   |-- syntax_highlight.py
```

```
|-- index.html.mako
|-- _posts
|   |-- 001 - post 1.markdown
|   |-- 002 - post 2.markdown
`-- _templates
    |-- atom.mako
    |-- base.mako
    |-- chronological.mako
    |-- footer.mako
    |-- header.mako
    |-- head.mako
    |-- permapage.mako
    |-- post_excerpt.mako
    |-- post.mako
    |-- rss.mako
    `-- site.mako
```

The basic building blocks of a Blogofile site are:

- **_config.py** - Your main Blogofile configuration file. See *Configuration File*
- **Templates** - Templates dynamically create pages on your site. `index.html.mako` along with the entire `_templates` directory are examples. See *Templates*
- **Posts** - Your blog posts, contained in the `_posts` directory. See *Posts*
- **Filters** - contained in the `_filters` directory, filters can process textual data like syntax highlighters, translators, swear word censors etc. See *Filters*
- **Controllers** - contained in the `_controllers` directory, controllers create dynamic sections of your site, like blogs. See *Controllers*

Any file or directory not starting with an underscore, and not ending in ".mako", are considered regular files (eg. `css/site.css` and `js/site.js`). These files are copied directly to your compiled site.

3.3 Building the Site

Now that you have an example site initialized, we can compile the source to create a functioning website.

Run the following to compile the source in the current directory:

```
blogofile build
```

Blogofile should run without printing anything to the screen. If this is the case, you know that it ran successfully. Inside the _site directory you have now built a complete website based on the source code in the current directory. You can now upload the contents of the _site directory to your webserver or you can test it out in the embedded webserver included with Blogofile:

```
blogofile serve 8080
```

Go to http://localhost:8080 to see the site served from the embedded webserver. You can quit the server by pressing `Control-C`.

3.4 Understanding the Build Process

When the Blogofile build process is invoked, it follows this conceptual order of events:

- A `_config.py` file is loaded with your custom settings. See *Configuration File*.
- If the blog feature is enabled (*blog.enabled*), the blog posts in the `_posts` directory are processed and made available to templates. See *Posts*.
- Filters in the `_filters` directory are made available to templates. See *Filters*.
- Files and sub-directories are recursively processed and copied over to the `_site` directory which becomes the compiled HTML version of the site:
  - If the filename ends in `.mako`, it is considered a page template. It is rendered via Mako, then copied to the `_site` directory stripped of the `.mako` extension. See *Templates*.
  - If the filename or directory starts with an underscore, it is ignored and not copied to the `_site` directory (other ignore patterns may be setup using *site.file_ignore_patterns* in `_config.py`.)
- Controllers from the `_controllers` directory are run to build dynamic sections of your site, for example, all of the blog features: permalinks, archives, categories etc. See *Controllers*.

3.4.1 Build Process Flowchart

Click for larger SVG view

CHAPTER

FOUR

CONFIGURATION FILE

Blogofile looks for a file called `_config.py` in the root of your source directory; this is your site's main configuration file. Blogofile tries to use sensible default values for anything you don't configure explicitly in this file. Although every site must have a `_config.py`, it can start out completely blank.

`_config.py` is just regular Python source code. If you don't know any Python, don't worry, there's actually very little you need to change in this file to get started.

4.1 Context of _config.py

`_config.py` is run within a context that is prepared by Blogofile before executing. This context includes the following objects:

- **controllers** - Settings for each controller (See *Controllers*).
- **filters** - Settings for each filter (See *Filters*).
- **site** - General Settings pertaining to your site, eg: site.url.

All of these are instances of the HierarchicalCache class. HierarchicalCache objects behave a bit differently than typical Python objects: accessed attributes that do not exist, do not raise an AttributeError. Instead, they instantiate the non-existing attribute as a nested HierarchicalCache object.

This style of configuration provides a seperate namespace for each feature of your blogofile site, and also allows for Blogofile to contain configuration settings for controllers or filters that may or may not be currently installed. For example, your `_config.py` might have the following setting for a photo gallery controller:

```
controllers.photo_gallery.albums.photos_per_page = 5
```

In the above example, `controllers`, `photo_gallery`, and `albums`, are all instances of HierarchicalCache. `photos_per_page` is an integer that is an attribute on the `albums` HierarchicalCache.

Because this setting is contained in a HierarchicalCache object, if the photo gallery controller is not installed, the setting will simply be ignored.

4.2 Site Configuration

In Blogofile, the "site" corresponds with the `_site` directory that blogofile builds. Even if your site is primarily used as a blog, think of the "site" as the parent of the blog. The site has it's own namespace within `_config.py` called `site`.

4.2.1 site.url

String

This is the root URL for your website. This is the URL that your blogofile site will be hosted at:

```
site.url = "http://www.xkcd.com"
```

4.2.2 site.file_ignore_patterns

List

This is a list of regular expressions that describe paths to ignore when processing your source directory. The most important one (and one you should not remove) is `".*/_.*"` which ignores all files and directories that start with an underscore (like `_config.py` and `_posts`):

```
site.file_ignore_patterns = [
# All files that start with an underscore
".*/_.*",
# Emacs temporary files
".*/#.*",
# Emacs/Vim temporary files
".*~$",
# Vim swap files
".*/\..*\.swp$",
# VCS directories
".*/\.(git|hg|svn|bzr)$",
# Git and Mercurial ignored files definitions
".*/.(git|hg)ignore$",
# CVS dir
".*/CVS$",
]
```

4.3 Blog Configuration

The core of Blogofile actually does not know what a blog is. Blogofile itself just provides a runtime environment for templates, controllers and filters. A Blogofile blog is actually built by creating a blog controller (see *Controllers*.) A default implementation of a blog controller is provided with the Blogofile `simple_blog` template and should be sufficient for most users.

All controllers in Blogofile have their own seperate namespace in `_config.py` under `controllers`. For convenience, you would usually reference the blog controller like so in `_config.py`:

```
blog = controllers.blog
```

4.3.1 blog.enabled

Boolean

This turns on/off the blog feature. Blogofile is obviously geared toward sites that have blogs, but you don't *need* to have one. If this is set to True, Blogofile requires several blog specific templates to exist in the `_templates` directory as described in *Template Environment*:

```
blog.enabled = True
```

4.3.2 blog.path

String

This is the path of the blog off of the *site.url*. For example, if *site.url* is `http://www.xkcd.com/stuff` and blog.path is `/blag` your full URL to your blog will be `http://www.xkcd.com/sfuff/blag`:

```
blog.path = "/blog"
```

4.3.3 blog.name

String

This is the name of your blog:

```
blog.name = "xkcd - The blag of the webcomic"
```

4.3.4 blog.description

String

This is a (short) description of your blog. Many RSS readers support/expect a description for feeds:

```
blog.description = "A Webcomic of Romance, Sarcasm, Math, and Language"
```

4.3.5 blog.timezone

String

This is the timezone that you normally post to your blog from:

```
blog.timezone = "US/Eastern"
```

You can see all of the appropriate values by running:

```
python -c "import pytz, pprint; pprint.pprint(pytz.all_timezones)" | less
```

4.3.6 blog.posts_per_page

Integer

This is the number of blog posts you want to display per page:

```
blog.posts_per_page = 5
```

4.3.7 blog.auto_permalink.enabled

Boolean

This turns on automatic permalink generation. If your post does not include a permalink field, then this allows for the automatic generation of the permalink:

```
blog.auto_permalink.enabled = True
```

4.3.8 blog.auto_permalink.path

String

This is the format that automatic permalinks should take on, starting with the path after the blog domain name. eg: `/blag/:year/:month/:day/:title` creates a permalink like `http://www.xkcd.com/blag/2009/08/18/post-one`:

```
blog.auto_permalink.path = ":blog_path/:year/:month/:day/:title"
```

Available replaceable items in the string:

- :blog_path - The root of the blog
- :year - The post year
- :month - The post month
- :day - The post day
- :title - The post title
- :uuid - sha hash based on title
- :filename - the filename of the post (minus extension)

4.3.9 blog.disqus.enabled

Boolean

Turns on/off Disqus comment system integration:

```
blog.disqus.enabled = False
```

4.3.10 blog.disqus.name

String

The Disqus website 'short name':

```
blog.disqus.name = "your_disqus_name"
```

4.3.11 blog.custom_index

Boolean

When you configure *blog.path*, Blogofile by default writes a chronological listing of the latest blog entries at that location. With this option you can turn that behaviour off and your index.html.mako file in that same location will be your own custom template:

```
blog.custom_index = False
```

4.3.12 blog.post_excerpts.enabled

Boolean

Post objects have a `.content` attribute that contains the full content of the blog post. Some blog authors choose to only show an excerpt of the post except for on the permalink page. If you turn this feature on, post objects will also have a `.excerpt` attribute that contains the first *blog.post_excerpts.word_length* words:

```
blog.post_excerpts.enabled = True
```

If you don't use post excerpts, you can turn this off to decrease render times.

4.3.13 blog.post_excerpts.word_length

Integer

The number of words to have in post excerpts:

```
blog.post_excerpts.word_length = 25
```

4.3.14 blog.pagination_dir

String

The name of the directory that contains more pages of posts than can be shown on the first page.

Defaults to `page`, as in `http://www.test.com/blog/page/4`:

```
blog.pagination_dir = "page"
```

4.3.15 blog.post_default_filters

Dictionary

This is a dictionary of file extensions to default filter chains (see *Filters*) to be applied to blog posts. A default filter chain is applied to a blog post only if no filter attribute is specified in the blog post YAML header:

```
blog.post_default_filters = {
    "markdown": "syntax_highlight, markdown",
    "textile": "syntax_highlight, textile",
    "org": "syntax_highlight, org",
    "rst": "syntax_highlight, rst",
    "html": "syntax_highlight"
}
```

4.4 Build Hooks

4.4.1 pre_build

Function

This is a function that gets run before the _site directory is built

4.4.2 post_build

Function

This is a function that gets run after the _site directory is built .. _config-post-build:

4.4.3 build_finally

Function

This is a function that gets run after the _site directory is built OR whenever a fatal error occurs. You could use this function to perform a cleanup function after building, or to notify you when a build fails.

CHAPTER

FIVE

TEMPLATES

Templates are at the very heart of Blogofile; they control every aspect of how the site is structured. Blogofile uses the Mako templating engine which has an active community and great documentation. Blogofile doesn't try to limit what you can do with your templates, you've got the full power of Mako so go ahead and use it.

Blogofile makes a distinction between two basic kinds of templates:

- **Page** templates
- **Reusable** templates

Page templates represent a single unique page (or URL) on your site. These are files somwhere in your source directory that end in `.mako` and never reside in a directory starting with an underscore. Page templates are rendered to HTML and copied to the `_site` directory in the same location where they reside in the source directory. Examples: an index page, a contact page, or an "about us" page.

Reusable templates are contained in the _templates directory. These are features that you want to include on many pages. Examples: headers, footers, sidebars, blog post layouts etc. Reusable templates do not represent any particular page (or URL) but are rather inherrited or included inside other templates or *Controllers* and usually reused on many diverse pages.

5.1 A Simple Example Using Just Mako

It would be redundant to describe all the things you can do with Mako when great documentation already exists, but a few simple examples of templates are in order.

The first thing a website needs is an index or 'home' page. Here's how you create one in blogofile:

In the root of your source directory create a file called `index.html.mako`:

```
<%inherit file="_templates/site.mako" />
This is the index page contents.
```

Well, there's not much in there, but that's by design. To effectively use Mako, you're going to want to use inheritance to allow you to abstract out the things that you want to repeat on every page (headers, footers, sidebars etc) from the things that don't repeat: the content of the page.

The `index.html.mako` inherits from a **reusable template** called `site.mako`. Create a file called `_templates/site.mako`:

```
<html>
  <body>
   ${self.header()}
    <div id="main_block">
      ${next.body()}
```

```
    </div>
    ${self.footer()}
  </body>
</html>
<%def name="header()">
  This is a header you want on every page
  <hr/>
</%def>
<%def name="footer()">
  <hr/>
  This is a footer you want on every page
</%def>
```

At the bottom of `site.mako` there are two `<%def>` blocks: header and footer. Think of these `<%def>` blocks as functions that can be reused multiple times. Each of these defs are referenced inside the `<html>` tag above (eg. `${self.header()}`). Referencing the def block simply deposits the contents of the `<def>` wherever it's referenced.

You could simply write the header and footer inline in the HTML and you'd still get the same effect of having them appear on every page that inherits from `site.mako`, however if you create them as `<%def>` blocks, you can redefine these blocks on child templates so that a different header or footer can appear on some pages while retaining the rest of the look and feel of the `site.mako` template.

One special reference is also made to `${next.body()}`. This deposits the contents of any child templates that inherit from this template. In our example, `index.html.mako` inherits from `site.mako`, so the text `this is the index page contents` is deposited inside the `<div id="main_block">` in the resulting HTML file which looks something like this:

```
<html>
  <body>
    This is a header you want on every page
    <hr/>
     <div id="main_block">
      This is the index page contents.
     </div>
    <hr/>
    This is a footer you want on every page
  </body>
</html>
```

5.2 Adding Blogofile Features To Our Templates

In the last section we introduced a simple template called `index.html.mako`. This template is the home page of our site, and so far only includes regular mako functionality. Now let's introduce some Blogofile action!

Let's say we want to include on our home page a list of the 5 most recent posts from our blog. As long as *blog.enabled* is turned on, each template can get access to our blog posts through a cache object called `bf`. We can modify our `index.html.mako` to get the list of recent posts:

```
<%inherit file="_templates/site.mako" />
Here's the five most recent posts from the blog:

<ul>
% for post in bf.config.blog.posts[:5]:
  <li><a href="${post.path}">${post.title}</a></li>
```

```
% endfor
</ul>
```

If you're familiar with for-loops in Python, this should look somewhat similar. We create an unordered list tag and inside that list we iterate over a special Blogofile object containing all of our posts. We limit ourselves to the first 5 posts by slicing the list of posts from 0 to 5.

Each post contains various metadata (see *Posts*) about the post. In this example we are interested in two things: the relative URL to the permalinked post as well as the title of the post. We create the anchor containing the relative URL `${post.path}` and we name the anchor the same as the post `${post.title}`. The rendered HTML file will now look something like this:

```
<html>
  <body>
    This is a header you want on every page
    <hr/>
     <div id="main_block">
      Here's the five most recent posts from the blog:
      <ul>
       <li><a href="/blog/2009/08/29/profit">Profit!</a></li>
       <li><a href="/blog/2009/08/29/halcyon-and-on-and-on">Halcyon and On and On</a></li>
       <li><a href="/blog/2009/08/29/were-on-a-roll">We're on a roll</a></li>
       <li><a href="/blog/2009/08/29/another-post">Another Post</a></li>
       <li><a href="/blog/2009/08/22/first-post">First Post!</a></li>
      </ul>
     </div>
    <hr/>
    This is a footer you want on every page
  </body>
</html>
```

5.3 Template Environment

In the last section we introduced a special Blogofile object called `bf`. This object is a gateway to all things related to Blogofile and is provided to all your templates.

You can also import it into your *Controllers* and *Filters*:

```
import blogofile_bf as bf
```

5.3.1 Blogofile modules

`bf` holds all of the core Blogofile modules, for example:

- `bf.util`
- `bf.config`
- `bf.writer`

5.3.2 Controller configuration

`bf` holds all the controller configuration, for example:

- `bf.controllers.blog.enabled`

- bf.controllers.blog.path

5.3.3 Filter configuration

bf holds all the filter configuration, for example:

- bf.filters.syntax_highlight.enabled
- bf.filters.syntax_highlight.style

5.3.4 Template context

When a template is being rendered, it's sometimes useful to be able to maintain a context available throughout the time that a given template is being rendered. If, for example, you are rendering a template called my_cool_template.mako which inherits from site.mako and includes sidebar.mako, a single context will be maintained that can be accessed from all three of those templates.

bf.template_context is a HierarchicalCache object and is available inside any template and you can put whatever data you want on it. The one peice of information that is included by default is bf.template_context.template_name which records the original template requested to be rendered. In the above example, this would be my_cool_template.mako.

CHAPTER

SIX

CONTROLLERS

Controllers are used when you want to create a whole chunk of your site dynamically everytime you compile your site. The best example of this is a blog. The whole purpose of a blog engine is to make it so you don't have to update 10 different things when you just want to make a post. Examples of controllers include:

- A sequence of blog posts listed in reverse chronological order paginated 5 posts per page.
- A blog post archiver to list blog posts in reverse chronological order listed by year and month.
- A blog post categorizer to list blog posts in reverse chronological order listed by category.
- An RSS/Atom feed generator for all posts, or for a single category.
- A permalink page for all blog posts.

All of these are pretty much a necessity for a blog engine, but none of these are included within the core of Blogofile itself. One of Blogofile's core principles is to remain light, configurable, and to make little assumption about how a user's site should behave. All of these blog specific tasks are relegated to a type of plugin system called controllers so that they can be tailored to each individual's tastes as well as leave room for entirely new types of controllers written by the user.

The simple_blog sources (which you can obtain by running `blogofile init simple_blog`) include all of these controllers in the `_controllers` directory. But let's look at an even simpler example for the purposes of this tutorial.

6.1 A Simple Controller

Suppose you wanted to create a simple photo gallery with a comments page for each photo. You don't want to have to create a new mako template for every picture you upload, so let's write a controller instead. The controller will be really simple: read all the photos in the photo directory and create a single page for each photo and allow comments on the photo using Disqus. While we're at it, let's also create an index page listing all the photos with a thumbnail and the name of the image.

First create the controller called `_controllers/photo_gallery.py`:

```
# A stupid little photo gallery for Blogofile.

# Read all the photos in the /photos directory and create a page for each along
# with Disqus comments.

import os
from blogofile.cache import bf

config = {"name"        : "Photo Gallery",
```

```
          "description" : "A very simplistic photo gallery, used as an example",
          "priority"    : 40.0}

photos_dir = os.path.join("demo","photo_gallery")

def run():
    photos = read_photos()
    write_pages(photos)
    write_photo_index(photos)

def read_photos():
    #This could be a lot more advanced, like supporting subfolders, creating
    #thumbnails, and even reading the Jpeg EXIF data for better titles and such.
    #This is kept simple for demonstration purposes.
    return [p for p in os.listdir(photos_dir) if p.lower().endswith(".jpg")]

def write_pages(photos):
    for photo in photos:
        bf.writer.materialize_template("photo.mako",
                (photos_dir,photo+".html"), {"photo":photo})

def write_photo_index(photos):
    bf.writer.materialize_template("photo_index.mako",
                (photos_dir,"index.html"), {"photos":photos})
```

When a controller is loaded, the first thing Blogofile looks for is a `run()` method to invoke. It never takes any arguments, each controller is expected to know what it's going to do of it's own accord.

In this example the `run()` method does all the work:

- It reads all the photos: `read_photos()`
- It creates a page for each photo: `write_pages()`
- It creates a single index page for all the photos: `write_photo_index()`

The `bf.writer.materialize_template` method is provided to make it easy to pass data to a template and have it written to disk inside the `_site` directory.

The `write_pages()` method references a reusable template residing in `_templates/photo.mako`:

```
<%inherit file="site.mako" />
<center><img src="/demo/photo_gallery/${photo}"/></center>
<div id="disqus_thread"></div>
<script type="text/javascript">
  var disqus_url = "${bf.config.site.url}/demo/photo_gallery/${photo}.html";
</script>
<script type="text/javascript"
   src="http://disqus.com/forums/${bf.config.blog.disqus.name}/embed.js"></script>
<noscript><a href="http://${bf.config.blog.disqus.name}.disqus.com/?url=ref">
    View the discussion thread.</a>
</noscript><a href="http://disqus.com" class="dsq-brlink">blog comments powered by
<span class="logo-disqus">Disqus</span></a>
```

The controller passes in a single variable: `photo`, which is the filename of the photo. In a more complete photo gallery, one might pass an object that held the EXIF data.

The `write_photo_index()` method references a reusable template residing in `_templates/photo_index.mako`:

```
<%inherit file="_templates/site.mako" />
My Photos:
<table>
% for photo in photos:
  <tr><td><a href="${photo}.html">
    <img src="${photo}" height="175"></a></td><td>${photo}</td></tr>
% endfor
</table>
```

The controller passes a single variable: `photos`, which is a sequence of all the photos filenames. In a more complete photo gallery, one might pass a sequence of objects that had references to the full jpg as well as a thumbnail and EXIF data.

This example is included in the blogofile.com sources and can also be viewed live.

6.2 Controller structure

Controllers can be single .py files inside the _controllers directory, as in the photo gallery example above, or they can be full python modules (Python modules are directories with a `__init__.py` file). This second method will let you split your controller among multiple files.

Controllers are always disabled by default, and must be explicitly turned on in your `_config.py`. For example, to enable the photo gallery example:

```
controllers.photo_gallery.enabled = True
```

Controllers have a standardized configuration protocol. All controllers define a dictionary called `config`. By default it contains the following values:

```
config = {"name"        : None,
          "description" : None,
          "author"      : None,
          "url"         : None,
          "priority"    : 50.0,
          "enabled"     : False}
```

These settings are as follows:

- name - The human friendly name for the controller.
- author - The name or group responsible for writing the controller.
- description - A brief description of what the controller does.
- url - The URL where the controller can be downloaded on the author's site.
- priority - The default priority to determine sequence of execution. This is optional, if not provided, it will default to 50. Controllers with higher priorities get run sooner than ones with lower priorities.

These are just the default settings, a controller author may provide as many configuration settings as he wants.

A user can override any configuration setting in their `_config.py`:

```
controllers.photo_gallery.albums.photos_per_page = 5
```

6.3 Controller Initialization

Controller's have an additional optional method called `init()`. Like the `run()` method, it doesn't take any arguments, it's expected that the controller knows how to initialize itself. The initialization is useful when you need to perform some preparation work before running the main controller. Typical use cases are where two controllers interact with each other and have cyclical dependencies on one another. With an initialization step, you can avoid chicken-or-the-egg problems between two controllers that require data from each other at runtime.

CHAPTER

SEVEN

POSTS

Posts are interpreted by the blog controller that you get when you instantiate the simple_blog; they have no particular meaning to the core runtime of Blogofile. If you wanted, you could reimplement the blog controller yourself and use whatever post formatting you wished. It's expected that most users will just use the default blog controller, so this Post documentation is here for convenience.

Blog posts go inside the **_posts** directory. Without the blog controller enabled, the **_posts** directory is ignored because it starts with an underscore.

Each post is a seperate file and you can name the files whatever you want, but it's suggested to prefix your posts with a number like `0001`, `0002` etc. so that when you look at the files in a directory they will be naturally ordered sequentially. It's important to realize that this order is not the same order that the blog controller uses in chronlogical listings. Instead it sorts the posts based on the date field described below.

7.1 An Example Post

Here's an example post:

```
---
categories: Category One, Category Two
date: 2009/08/18 13:09:00
permalink: http://www.blogofile.com/2009/08/18/first-post
title: First Post
---
This is the first post
```

The post is divided into two parts, the YAML header and the post content.

You can see more examples of Blogofile posts on the project site.

7.2 YAML Header

The YAML portion is between the two `---` lines, and it describes all of the metadata for the post. You can define as many fields as you like, but there are some names that are reserved for general purpose use:

- **title** A one-line free-form title for the post.
- **date** The date that the post was originally created. (year/month/day hour:minute:second).
- **updated** The date that the post was last updated. (year/month/day hour:minute:second).
- **categories** A list of categories that the post pertains to, each seperated by commas. You don't have to configure the categories beforehand, you are defining them right here.

- **tags** A list of tags that the post pertains to, each seperated by commas.
- **permalink** The full permanent URL for this post. This is optional, one will be generated automatically if left blank. (see *blog.auto_permalink.path*)
- **filters** The filter chain to run on the post content. (see *Filters*)
- **filter** A synonym for filters. (see *Filters*)
- **guid** A unique hash for the post, if not provided it is assumed that the permalink is the guid.
- **author** The name of the author of the post.
- **draft** If 'true' or 'True', the post is considered to be only a draft and not to be published. A permalink will be generated for the post, but the post will not show up in indexes or RSS feeds. You would have to know the full permalink to ever see the page.
- **source** Reserved internally.
- **yaml** Reserved internally.
- **content** Reserved internally.
- **filename** Reserved internally.

This list is also defined in the blogofile source code under `blogofile.post.reserved_field_names` and can be accessed as a dictionary at runtime.

7.3 Post Content

The post content is written using a markup language, currently Blogofile supports several to choose from:

- Markdown (files end in .markdown)
- Textile (files end in .textile)
- reStructuredText (files end in .rst)
- or plain old HTML (files end in .html by convention, but if it's not one of the above, posts default to HTML anyway)

Adding your own markup formats is easy, you just implement it as a filter (see *Filters*)

The content of the post goes directly after the YAML portion and uses whatever markup language is indicated by the file extension of the post file.

7.4 Referencing posts in templates

All the posts are stored in a cache object called `bf`. This object is exposed to all templates and you can reference it directly with `${bf.config.blog.posts}`. They are ordered sequentially by date. See *Adding Blogofile Features To Our Templates* for an example.

CHAPTER

EIGHT

FILTERS

Filters are Blogofile's text processor plugin system. They create callable functions in templates and blog posts that can perform manipulation on a block of text. Ideas for filters:

- Markup languages.
- A code syntax highlighter.
- A flash video plugin helper.
- A swear word filter.
- A foreign language translator.

8.1 A Simple Filter

Here's a swear word filter that replaces nasty words with the word `kitten`. The file is called `_filters/playnice.py`:

```
#Replace objectionable language with kittens.
#This is just an example, it's far from exhaustive.
import re

seven_words = re.compile(
  r"\Wfrak\W|\Wsmeg\W|\Wjoojooflop\W|\Wswut\W|\Wshazbot\W|\Wdoh\W|\Wgorram\W|\Wbelgium\W",
  re.IGNORECASE)

def run(content):
    return seven_words.sub(" kitten ", content)
```

This filter (once it's in your `_filters` directory) is available to all templates and blog posts.

8.2 Filter Chains

Filters can be chained together, one after the other, so that you can perform multiple text transformations on a peice of text.

Suppose you have the following filters in your `_filters` directory:

- **markdown.py** - Transforms Markdown formatted text into HTML.
- **playnice.py** - The swear word filter above.
- **syntax_highlight.py** - A code syntax highlighter

A filter chain of `markdown, playnice, syntax_highlight` will apply those three filters (seperated by commas) in the order given.

8.3 Using Filters in a Template

The filter module provides the method called `run_chain`. You can use this directly in your mako templates:

```
The following text is filtered:

${bf.filter.run_chain('playnice, syntax_highlight', 'some shazbot text')}
```

However, it's kind of a pain to always wrap the text you want to filter in that function call. Writing a mako `<%def>` block can create some nice syntactic sugar for us. Define the following in your base template so that all templates that inherit from it can benefit from it:

```
<%def name="filter(chain)">
 ${bf.filter.run_chain(chain, capture(caller.body))}
</%def>
```

How to use it in a template:

```
<%self:filter chain="playnice, syntax_highlight">Belgium: Less offensive
  words have been created in the many languages of the galaxy, such as
  joojooflop, swut and Holy Zarquon's Singing Fish. </%self:filter>
```

All the text between the `<%self:filter>` start and end tags is filtered by the specified filter chain.

8.4 Using Filters in a Blog Post

Filter chains can be applied to blog posts in the post YAML:

```
---
date: 2009/12/01 11:17:00
permalink: http://www.blogofile.com/whatever
title: A markdown formatted test post
filter: markdown, playnice
---
This is a **markdown** formatted post with all the frak words filtered.
```

Filters on blog posts are applied to the entire blog post; you cannot apply a filter to only a portion of the text like you can with templates. However, there is nothing preventing you from writing a filter that looks for special syntax in your posts and filters selectively (the syntax_highlight filter from simple_blog does exactly this). This allows for more end user customizability.

If no filter is specified for your post, Blogofile looks at a config option called *blog.post_default_filters* which maps the file extension of the post file to a filter chain. Defaults include `markdown` and `textile`.

You can turn off all filters for the post, including the default ones, by specifing a filter chain of `none`.

8.5 Filter structure

Filters can be single .py files inside the _filters directory, as in the `playnice.py` example above, or they can be full python modules (Python modules are directories with a `__init__.py` file). This second method will let you split your filters among multiple files.

Filters have a standardized configuration protocol. All filters define a dictionary called `config`. By default it contains the following values:

```
config = {"name"        : None,
          "description" : None,
          "author"      : None,
          "url"         : None}
```

These settings are as follows:

- name - The human friendly name for the controller.
- author - The name or group responsible for writing the controller.
- description - A brief description of what the controller does.
- url - The URL where the controller can be downloaded on the authors site.

These are just the default settings, a filter author may provide as many configuration settings as he wants.

A user can override any configuration setting in their `_config.py`:

```
filters.playnice.zealous_and_vigorous_parsing = True
```

CHAPTER

NINE

INTEGRATION WITH VERSION CONTROL

9.1 You Want Version Control

You might not know it yet, but you want your blog under a Version Control System (VCS). Consider the benefits:

- Regular and complete backups occur whenever you push your changes to another server.
- The ability to bring back any version of your site (or any single page) from history.
- The ability to work from any computer, without getting worried if you're working on the latest version or not.
- Automatic Deployment.

Automatic what? Even if you're a veteran to VCS, you may not realize that a VCS can do a lot more for you than just provide a place for you to dump your files. You can have your favorite VCS build and deploy your Blogofile based site for you everytime you commit new changes.

So why are you procrastinating? Get git.

9.2 Automatic Deployment in Git

You need to have the origin server (the place that you 'git push' to) be the same server that hosts your website for this example to work. [1]

On the server, checkout the project:

```
git clone /path/to/your_repo.git /path/to/checkout_place
```

Create a new `post-recieve` hook in your git repo by creating the file `/path/to/your_repo.git/hooks/post-receive`:

```
#!/bin/sh

#Rebuild the blog
unset GIT_DIR
cd /path/to/checkout_place
git pull
blogofile build
```

[1] If you deploy to a different server than the one hosting your git repository, you could just craft your own rsync or FTP command and put it at the bottom of the post-receive hook to deploy somewhere else. But that's beyond the scope of this document.

Configure your webserver to host your website out of `/path/to/checkout_place/_site`.

Now whenever you `git push` to your webhost, your webserver should get automatically rebuilt. If Blogofile outputs any errors, you'll see them on your screen.

9.3 Other VCS solutions

Most VCS should have support for a post recieve hook. If you create something cool in your own VCS of choice, let the blogofile discussion group know and we'll add it to this document.

OJ wrote up how to do something similar with mercurial.

CHAPTER

TEN

MIGRATING EXISTING BLOGS TO BLOGOFILE

Unless you're starting a brand new blog from scratch, you're probably going to want to migrate an existing blog to Blogofile. When migrating, you have to consider several things:

- Migrating existing blog posts.
- Migrating existing blog comments.
- Migrating permalinks and other search engine indexed URLs.

10.1 Wordpress

10.1.1 Comments

Before you bring your Wordpress blog offline, install the Disqus wordpress plugin. With this plugin, you can export all your comments from your wordpress database offsite into your Disqus account.

In your blogofile config file, set the *blog.disqus.enabled* and *blog.disqus.name* settings appropriately.

10.1.2 Posts

Download the converter script:

- wordpress2blogofile.py

Install SQL Alchemy:

```
easy_install sqlalchemy
```

If your database is MySQL based, you'll also need to download MySQLdb, which you can apt-get on Ubuntu:

```
sudo apt-get install python-mysqldb
```

If you're using some other database, install the appropriate DBAPI.

Edit `wordpress_schema.py`:

- `table_prefix` should be the same as you setup in wordpress, (or blank "" if none)
- `db_conn` point to your database server. The example is for a MySQL hosted database, but see the SQL Alchemy docs if you're using something else.

In a clean directory run the export script:

```
python wordpress2blogofile.py
```

If everything worked, you should now have a `_posts` directory containing valid Blogofile format posts which you can copy to your blogofile directory.

10.1.3 Permalinks

You're probably going to want to retain the exact same permalinks that your wordpress blog had. If you've been blogging for long, Google has inevitably indexed your blog posts and people may have linked to you; you don't want to change the permalink URLs for your posts.

The converter script should transfer over the permalinks directly into the *YAML Header* of the blog post. You may also want to configure the *blog.auto_permalink.path* setting in your configuration file to create the same style of permalink that you're using in wordpress.

10.2 Moveable Type

to be written.