engineer Documentation

Release 0.5.1.dev130

Tyler Butler <tyler@tylerbutler.com>

May 26, 2017

Contents

Note: Are you looking for documentation on the pre-release version of Engineer? If so, you can find them here: https://engineer.readthedocs.org/en/latest/.

- The current release version of Engineer is version 0.5.1.
- This documentation is for version 0.5.1.dev130.

What's New in Version 0.5.1?

- Atom feeds
- Jinja2 syntax support in post content
- Simpler way to include images in posts
- Support for *Markdown Lazy Links*
- Lots and lots of bug fixes

There's more! See the full *Release Notes* for details.

At its core, Engineer is a static website generator. In other words, Engineer let's you build a website from a bunch of files - articles written in Markdown, templates, and other stuff - and outputs *another* bunch of files - HTML, mostly - that you can then copy wherever you want. It has some very nice *Features* that will make you happy, but it's *not for everybody*.

Engineer was inspired by Brent Simmons, Marco Arment's Second Crack, Jekyll, Octopress, and Hyde.

Note: The Engineer documentation is a work in progress. It is by-and-large up-to-date and the most relevant sections are complete, but some of the more 'advanced' sections are not yet complete.

CHAPTER 1

Bugs and Feature Roadmap

If you find any bugs in Engineer please file an issue in the Github issue tracker (or fork and fix it yourself and send me a pull request). Feature ideas and other feedback are welcome as well!

CHAPTER 2

Narrative Documentation

Introduction

Overview

At its core, Engineer is a static website generator. In other words, Engineer let's you build a website from a bunch of files - articles written in Markdown, templates, and other stuff - and outputs *another* bunch of files - HTML, mostly - that you can then copy wherever you want.

Engineer was inspired by Brent Simmons, Marco Arment's Second Crack, Jekyll, Octopress, and Hyde.

Features

Write posts from anywhere

Posts can be written/edited in Markdown and stored/synchronized using Dropbox or another file synchronization solution.

Preview your site locally

Engineer includes a small *development web server* that you can use to preview your site locally without deploying anywhere.

Manage your site remotely

Even baked sites need a little management, and many existing static generators require you to load up the terminal and execute a command to rebuild your site. Engineer lets you *do that* of course, but also provides *Emma*, a built-in mini management site (optional) that lets you do most of the common management tasks remotely.

Themes make it easy to change your site's appearance

Themes provide flexibility in the site look and feel without starting from scratch or rewriting a bunch of content. You can write your own *Themes* as well.

Use LESS instead of CSS

Engineer lets you use LESS instead of CSS if you'd like. LESS can either be preprocessed on the server (requires that lessc be installed on non-Windows systems) or processed client-side using less.js.

It's fast

Engineer outputs content quickly (and I'm working to make it *even faster*), and because the output content is completely static, it is blazingly fast to serve, scales up very well, and is completely independent of any specific web server or technology. Once generated, you can copy your site *anywhere* and use any web server you like. In addition, Engineer can optimize your JavaScript and CSS/LESS to minimize their size. Engineer is all about speed.

Caveats

Despite all of these great features, there are some things that you might *not* like:

Dynamic things require a bit more work

Static sites can feel limited if you're accustomed to doing something super-dynamic every time a page is loaded. Most of these things can be handled using either client-side JavaScript (e.g. timeago.js) or clever uses of the Jinja 2 template system (see the navigation highlighting functionality in Engineer itself for an example of things that can be done).

Might not fit your site's needs

If you have a lot of one-off pages (*Template Pages* or other such things) then managing them can get a bit cumbersome. Engineer really excels when a majority of your site's content has a similar look and feel and you can leverage the *Metadata* for a majority of your content. Engineer isn't limited to blogs, per se, but it does make some assumptions that most of your content comes in the form of articles.

Only supports Markdown and Jinja 2

While ideally this will not always be true, currently Engineer requires your posts be written in Markdown and any templates you create be written in Jinja 2. This may change in the future, but for now you have to use those two languages.

Engineer is not a CMS

If you're looking for a full-blown content management system, then... keep looking. Engineer is decidedly not what you want. Engineer operates on the basic principle that your content is stored in text files with minimal metadata in the files themselves, so if you're looking for rich URL management, image/file manipulation capabilities, etc., Engineer will make you very sad. It's not designed to do that stuff.

Components

`engineer`

Engineer is primarily controlled by a command-line program aptly called *engineer*. It's used to build sites, configure *Emma*, start the *development server*, etc.

Theme Infrastructure

Engineer exposes a basic infrastructure and API that lets you create your own themes or use themes that others have created.

Plugin Architecture

Engineer provides a set of *Included Plugins* plus a way to *create your own*.

Requirements and Dependencies

Engineer requires Python 2.7+ and runs on Linux (Ubuntu and CentOS have been tested) and Windows. Chances are it will run on most platforms that Python and the Python packages Engineer depend on support, though exhaustive tests have not been run.

Engineer *has not* been tested on Python 3, and almost certainly will not work as-is since I have been a bit sloppy in my use of Python constructs that are deprecated in Python 3.

All relevant dependencies except Python itself will be installed when you *install Engineer*. The complete set of packages Engineer depends on is as follows:

Installation

Installing Using Pip

Installing Engineer is easy using pip. Simply run the following command:

```
pip install engineer
```

This will install the most recent released version of Engineer, which is version 0.5.1.

Installing from Source

Installing the Release Version from Source

If you'd prefer to install the current release version of Engineer (v 0.5.1) directly from the source, you have a couple of options. First, you can install directly from the GitHub repository using the following command:

```
pip install -e git+https://github.com/tylerbutler/engineer.git#egg=engineer
```

This will check out the latest files from the *master* (release) branch GitHub directly and install the package and all dependencies. Of course, you can also fork the repository and check out your own copy using the same approach.

Alternatively, you can download the source, unzip/untar it somewhere on your local hard drive, then run `setup.py`:

```
python setup.py install
```

Installing the In-Development Version from Source

If you're looking to install the in-development version of Engineer, then you can use the same methods covered above. Using pip, the command must be changed slightly:

```
pip install -e git+https://github.com/tylerbutler/engineer.git@dev#egg=engineer
```

If you download the Engineer source or clone the repository yourself, make sure you get the *dev* branch contents.

Note: If you install Engineer from source using either of these methods, you should ensure you're looking at the most recent version of this documentation that corresponds to the in-development version of Engineer. You can find that version of the documentation at https://engineer.readthedocs.org/en/latest/.

Creating a New Site

After installation, you can use the *engineer init* command to initialize a new site with some sample content and config files. Check the *Engineer Commandline* reference for more details about all the commands available, or read *Getting Started* if you're new to Engineer.

Upgrading to Engineer 0.5.0

Engineer 0.5.0 is released with a new version of *setuptools*. Due to some pretty big changes there, including the recombining of the *distribute* forked project with the main *setuptools* project, you may get an error when you try to upgrade Engineer in the standard way with `pip install -U engineer`. It may look something like this:

```
pip install -U engineer

Downloading/unpacking engineer from https://pypi.python.org/packages/source/e/
↪engineer/engineer-0.5.0.zip#md5=a1bb4061419a5430b91ae597032c801f
Downloading engineer-0.5.0.zip (3.5MB): 3.5MB downloaded
Running setup.py egg_info for package engineer

The required version of setuptools (>=2.1) is not available,
and can't be installed while this script is running. Please
install a more recent version first, using
'easy_install -U setuptools'.

(Currently using setuptools 0.6c11 (c:\users\tyler\.virtualenvs\engineer\lib\site-
↪packages\setuptools-0.6c11-py2.7.egg))
```

Fortunately, there are a few ways around this. First, you should upgrade *pip* and *setuptools*. There are details on how to do this on the pip website, but basically it boils down to running this command:

```
python -m pip install -U pip
```

Once pip is upgraded, then you can use it to upgrade setuptools itself:

```
pip install -U setuptools
```

Once *that's* done, you should be able to upgrade Engineer itself like so:

```
pip install -U engineer
```

Note that if you're using virtualenv, you may need to upgrade pip and setuptools in your virtualenv *as well as* the 'global' (outside the virtualenv) versions.

If for some reason these steps don't work, I suggest downloading get-pip.py, running it using `python get-pip.py`, then deleting and recreating any virtualenvs you're using for Engineer. Hopefully it won't come to this, though. The steps above should be all that's needed.

Getting Started

Engineer's *init* command can be used to create a sample Engineer site in a matter of seconds. The steps below will walk through that process. You can also look at the source for tylerbutler.com to get more ideas of what's possible with Engineer.

Your First Engineer Site

After you've installed Engineer, you can use the **`engineer`** command at the command line to interact with it. The *engineer init* command will create a basic folder structure for you in a directory of your choosing, and using that command is a good place to start if you're new to Engineer. Open up a terminal and type:

```
PS C:\> mkdir my-engineer-site
PS C:\> cd my-engineer-site
PS C:\my-engineer-site> engineer init
Initialization complete.
PS C:\my-engineer-site> ls

    Directory: C:\my-engineer-site

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----           4/7/2012   9:29 PM            archives
d----           4/7/2012   9:29 PM            posts
d----           4/7/2012   9:29 PM            templates
-a---           4/7/2012   9:29 PM        153 config.yaml
-a---           4/7/2012   9:29 PM         30 debug.yaml
-a---           4/7/2012   9:29 PM         35 oleb.yaml

PS C:\my-engineer-site>
```

Building the Site

Now you have a basic Engineer site, along with some sample content. We'll go over the details of what the files and folders in the site folder are used for, but for now, let's build the sample site:

```
PS C:\my-engineer-site> engineer build

Loading configuration from C:\my-engineer-site\config.yaml.
Found 8 new posts and loaded 0 from the cache.
Output new or modified post 'Engineer Documentation'.
Output new or modified post 'Welcome'.
Output new or modified post 'What's Next?'.
Output new or modified post 'Theme Style Preview'.
Output new or modified post 'Markdown Tutorial'.
Output new or modified post 'Post from 2011'.

Site: 'Engineer Site' output to C:\my-engineer-site\output.
Posts: 6 (6 new or updated)
Post rollup pages: 2 (5 posts per page)
Template pages: 2
Tag pages: 7
125 new items, 0 modified items, and 0 deleted items.

Full build log at C:\my-engineer-site\logs\build.log.
```

A few seconds after typing **`engineer build`** you should see some output similar to the above. The last few lines provide a summary of the overall build. In this case, there were four new posts, a rollup page, two template pages, and seven tag pages output. A total of 125 new files were output - that count includes static files such as JavaScript and CSS. For fun, let's see what happens if we run the build command again immediately:

```
PS C:\my-engineer-site> engineer build

Loading configuration from C:\my-engineer-site\config.yaml.
Found 0 new posts and loaded 8 from the cache.

Site: 'Engineer Site' output to C:\my-engineer-site\output.
Posts: 6 (0 new or updated)
Post rollup pages: 2 (5 posts per page)
Template pages: 2
Tag pages: 7
0 new items, 0 modified items, and 0 deleted items.

Full build log at C:\my-engineer-site\logs\build.log.
```

You'll notice that the output is slightly different. In this case, the same number of posts, template pages, tag pages, etc. were output, but Engineer didn't end up changing any output files. This is because Engineer recognized that there weren't any changes to the source files that required outputting content.

Seeing What Your Site Looks Like

Now let's see what that site we just built actually looks like! We can use the built-in development server to do that:

```
PS C:\my-engineer-site> engineer serve

Loading configuration from C:\my-engineer-site\config.yaml.
Loading configuration from C:\my-engineer-site\config.yaml.
Bottle server starting up (using WSGIRefServer())...
Listening on http://localhost:8000/
Hit Ctrl-C to quit.
```

If you visit http://localhost:8000/ you'll see the output of the build process just as it would look if you copied the output folder to another web server. You can click around the site as much as you'd like. When you're done, you can shut down the development server by pressing `Ctrl-C`.

Now let's see what happens if we make a change to the site. Let's publish one of the draft posts in the `posts` folder. Open `(d)2012-03-18-test-post.md` in a text editor (any one will do) and you should see something like this:

```
title: Test Post
timestamp: 05:51 PM Sunday, March 18, 2012 UTC
status: draft
slug: test-post

---

This is a test post.
```

Change the line that says `status: draft` to read `status: published` instead and save the file. Then do another build:

```
PS C:\my-engineer-site> engineer build

Loading configuration from C:\my-engineer-site\config.yaml.
Found 1 new posts and loaded 7 from the cache.
Output new or modified post 'Test Post'.

Site: 'Engineer Site' output to C:\my-engineer-site\output.
Posts: 7 (1 new or updated)
```

```
Post rollup pages: 2 (5 posts per page)
Template pages: 2
Tag pages: 7
3 new items, 5 modified items, and 0 deleted items.

Full build log at C:\my-engineer-site\logs\build.log.
```

In this case, we see that there were several new files and folders created as well as some updates ones. Now use **`engineer serve`** to see what the site looks like. You should see the new post that we just published. Finally, try deleting a file in the `posts` folder, rebuilding, and see what happens...

While the sample site serves as a good starting point and a great way to familiarize yourself with the Engineer command line interface, it's probably not what you want your site to look like. Let's look at the files and folders in the site directory to see what we might want to change.

See also:

Engineer command reference

File System Structure

The file system in `C:\my-engineer-site\` should look something like this:

```
/my-engineer-site
   /_cache
   /archives
   /output
   /posts
   /templates
   - base.yaml
   - config.yaml
   - debug.yaml
```

You can ignore the `_cache` folder. It's just used by Engineer to improve performance. You could even delete it if you wanted; Engineer would simply recreate if needed. The `.yaml` files are used for configuration - there are a couple of different ones available so the same site can be generated in different ways out output to different locations.

The `archives` and `posts` folders contain *Posts* for the site. The `templates` folder contains *Templates*, including *Template Pages*, and the `output` folder contains - you guessed it! - the output content of your site after it's built by Engineer.

As you can see, each of these folders contains content used to build out the site. For more information about each of these things, see the relevant topic guides.

See also:

The following topic guides have specific information about the major components used in Engineer:

- *Settings Files*
- *Post*
- *Templates*
- *Template Pages*
- *Themes*

Settings Files

Engineer is configured using a simple settings file (or several settings files if you so desire). The file should contain the desired site settings in YAML. A typical settings file looks like this:

```
SITE_TITLE: Engineer Site
HOME_URL: '/'
SITE_URL: http://localhost:8080

PUBLISH_DRAFTS: no
POST_DIR:
  - posts
  - archives

THEME_SETTINGS:
  typekit_id: vty2qol

POST_TIMEZONE: 'America/Los_Angeles'
```

All top-level Engineer settings are in all caps with underscores separating words. Themes or other plugins may have their own specific settings that do not follow this convention. A comprehensive list of all the settings is below, but in practice only a few of them are needed.

Content Location Settings

These settings control the location on the local file system where Engineer should either look for or output files.

class `engineer.conf.`**`EngineerConfiguration`**

`SETTINGS_DIR`
Default: path to folder containing settings file

The path to the directory containing the settings file used. This is usually set automatically based on the location of the settings file used.

`CONTENT_DIR`
Default: `SETTINGS_DIR/content`

The path to the directory that contains any *Raw Content* for the site. Raw content includes things like favicons, `robots.txt` files, etc. Raw content is always processed last in *The Build Pipeline*, so anything in this folder will overwrite any automatically generated content.

`POST_DIR`
Default: `[SETTINGS_DIR/posts]`

A list of paths that contain *Posts* for the site. You can specify a single path here or multiple paths. When specifying multiple paths the files will always be processed in their original directory.

If you wish to include all subdirectories within a path, simply add a $\star$ to the end of the path. By default, however, Engineer will only include posts in the directory specified.

See also:

The Build Pipeline

Changed in version 0.5.0: Now supports including subdirectories within a post directory.

OUTPUT_DIR
Default: `SETTINGS_DIR/output`

The path that the generated site should be output to. By using multiple settings files, each with a different `OUTPUT_DIR` setting, it is easy to push out multiple copies of a site to different locations without changing anything in the source files.

OUTPUT_DIR_IGNORE
Default: `['.git', '.gitignore']`

A list of paths that should be completely ignored in the target directory when outputting the site. Ordinarily, Engineer will overwrite any files/folders in the output target that are not generated by the build process. In some cases this is not appropriate, such as when you are building a site and placing the built files in a git repository. The default setting ignores the `.git` directory in the target as well as the `.gitignore` file, so its contents will not be overwritten by the build process.

Paths should either be absolute or relative to the folder in which the settings file is located. They can either be to individual files or to folders. If an ignored folder contains no files, however, it will not be properly ignored in all cases. Thus you should ensure any ignored folders contain at least one file.

Tip: If you set the *OUTPUT_DIR_IGNORE* setting expicitly, the defaults will be overwritten completely, so you should add `.git` and/or `.gitignore` explicitly if you want them to be ignored.

New in version 0.5.0.

TEMPLATE_DIR
Default: `SETTINGS_DIR/templates`

The path to the directory containing site-specific *Templates*, including templates used for themes.

See also:

Themes

TEMPLATE_PAGE_DIR
Default: `TEMPLATE_DIR/pages`

The path to the directory containing *Template Pages*. These can be outside your standard *TEMPLATE_DIR* if you wish; for example, you may set this to be `/pages` and place your template pages in the root of your site content directory rather than with other templates.

See also:

Template Pages, *Themes*

CACHE_DIR
Default: `SETTINGS_DIR/_cache/<settings file name>/`

The path Engineer should place its caches. This location should be unique per config, and by default varies based on the name of the settings file used. **In general you should not need to modify this.**

CACHE_FILE
Default: `CACHE_DIR/engineer.cache`

The Engineer cache file location. **In general you should not need to modify this.**

OUTPUT_CACHE_DIR
Default: `CACHE_DIR/output_cache`

The Engineer output cache directory. **In general you should not need to modify this.**

JINJA_CACHE_DIR
Default: `CACHE_DIR/jinja_cache`

The Jinja cache directory. **In general you should not need to modify this.**

Deprecated since version 0.5.0: This setting is no longer exposed as of version 0.5.0.

BUILD_STATS_FILE
Default: `CACHE_DIR/build_stats.cache`

The Engineer build stats cache file location. **In general you should not need to modify this.**

LOG_DIR
Default: `SETTINGS_DIR/logs`

The Engineer log directory. All build logs will be stored in this directory. **In general you should not need to modify this.**

LOG_FILE
Default: `LOG_DIR/build.log`

TODO

Site Settings

class `engineer.conf.`**EngineerConfiguration**

SITE_TITLE
Default: `'SITE_TITLE'`

The title of your site. Where this text appears depends on your theme, but you should always set it since it usually appears very prominently, such as in the main header.

SITE_URL
Default: `'SITE_URL'`

The absolute URL to your site. For example, `http://tylerbutler.com/`. This is used to generate some links in your site, so it should be accurate. In general, Engineer generates relative URLs for use internally, but there are some cases, such as the Atom and RSS feeds, that require the absolute URL.

HOME_URL
Default: `'/'`

The root URL to your site. By default this is set to `/` which assumes your Engineer site will live at the root of a domain. However, if you're putting your site at http://example.com/blog, for example, you would set this to `/blog` so Engineer would generate URLs correctly for you.

STATIC_URL
Default: `HOME_URL/static`

The relative URL to your static content such as JavaScript and CSS files.

PERMALINK_STYLE
Default: `pretty`

A format string that controls how links to your posts are formatted. You can use one of the built-in permalink styles (described below) or provide a custom one. Permalink format strings should use standard Python string formatting. The following named parameters are available for you to use in your format string:

year The year portion of the post's timestamp as an integer.

month The month portion of the post's timestamp as string - includes a leading zero if needed.

day The day portion of the post's timestamp as a string - includes a leading zero if needed.

i_month The month portion of the post's timestamp as an integer.

i_day The day portion of the post's timestamp as an integer.

title (or slug) The post's slug.

timestamp The post's timestamp as a datetime.

post The post object itself.

Using the post and timestamp parameters you can create complex permalink styles, but for most purposes the year/month/day/slug convenience parameters are enough and simpler to use.

Built-in styles:

Engineer also provides some styles you can use directly. Simply use the name of the style below instead of defining your own.

Style	Format String
`pretty`	`{year}/{month}/{title}/`
`fulldate`	`{year}/{month}/{day}/{title}/`
`slugdate`	`{year}/{month}/{day}/{title}.html`

Changed in version 0.5.0: The default value for this setting changed to `pretty` in version 0.5.0. The previous default value was `fulldate`, so you can set it manually if you wish to retain the previous behavior.

SITE_AUTHOR
Default: `None`

The name of the primary author of your site. May be used by themes.

ROLLUP_PAGE_SIZE
Default: `5`

This setting controls how many posts are displayed on a rollup page such as the main site home page.

URLS
Default: n/a

TODO

Atom/RSS Feed Settings

class `engineer.conf.`**EngineerConfiguration**

FEED_TITLE
Default: `SITE_TITLE Feed`

The title of the site's Atom feed.

FEED_ITEM_LIMIT
Default: `ROLLUP_PAGE_SIZE`

Controls how many posts are listed in the site Atom feed.

FEED_DESCRIPTION
Default: `The FEED_ITEM_LIMIT most recent posts from SITE_TITLE.`

The description of the site's Atom feed.

FEED_URL
Default: `HOME_URL/feeds/atom.xml`

The URL of the site's Atom feed. This only affects the links to the feed that are output in templates using *urlname('feed')*. The feed itself will still be written out to `HOME_URL/feeds/atom.xml`, so you can configure a Feedburner URL, for example, as your feed URL, and then point Feedburner to `HOME_URL/feeds/atom.xml`.

Changed in version 0.5.0: The default feed format changed to Atom in Engineer 0.5.0. A feed in the RSS format is still generated and output to `HOME_URL/feeds/rss.xml`, but all the default feed-related settings point to the Atom formatted feed.

Theme Settings

See also:

Themes

class `engineer.conf.`**EngineerConfiguration**

THEME
Default: `dark_rainbow`

The *ID* of the theme to be used for the site.

THEME_DIRS
Default: `None`

A list of paths that contain *Themes* for the site. You can specify a single path here or multiple paths.

Note: You do not need to use this setting if custom themes are found inside the `themes` folder within the site's folder.

New in version 0.2.3.

THEME_SETTINGS
Default: `{}`

Any theme-specific settings. This is a dictionary of settings that the theme in use will use. What is appropriate for this setting differs based on the theme.

THEME_FINDERS
Default: `['engineer.themes.finders.ThemeDirsFinder', 'engineer.themes.finders.SiteFinder', 'engineer.themes.finders.PluginFinder', 'engineer.themes.finders.DefaultFinder']`

TODO

Changed in version 0.2.3.

Preprocessor/Compressor Settings

See also:

The Build Pipeline

class `engineer.conf.`**`EngineerConfiguration`**

`COMPRESSOR_ENABLED`
Default: `True`

If `True`, JavaScript and CSS files will be minified as part of the site generation process.

`COMPRESSOR_FILE_EXTENSIONS`
Default: `['js', 'css']`

The file extensions that should be minified.

Note: This setting shouldn't be used at this point - it's there because there are plans to make the minification process more configurable.

`PREPROCESS_LESS`
Default: `True`

If `True`, LESS files referenced in templates will be processed into CSS files (which will then be minified if needed) as part of the site generation process.

`LESS_PREPROCESSOR`
Default: bundled *less.js-windows* compiler on Windows, **`lessc`** elsewhere

If you want to use another LESS processor, or you need to specify a path to **`lessc`**, you can use this setting. On Windows a less compiler is bundled, but on other platforms you'll need to download and install less and lessc yourself. There is information about how to do that at lesscss.org.

Miscellaneous Settings

class `engineer.conf.`**`EngineerConfiguration`**

`ACTIVE_NAV_CLASS`
Default: `current`

When set, active navigation links (output using the *navigation_link* macro) will have the specified class.

Note: This value can still be overridden in individual calls to `navigation_link` by passing an `active_class` parameter.

`DEBUG`
Default: `False`

This flag is used to designate a site is in debug mode. Templates or other Engineer code might output content slightly differently in debug mode to provide more details about the rendering process. This should *always* be set to `False` when building a site for production.

Note: This setting is different than the *`--verbose`* option passed into the Engineer commandline. The `--verbose` option only changes the level of output at the command line. The `DEBUG` setting can be used to change up actual template rendering or code processing.

NORMALIZE_INPUT_FILES
Deprecated since version 0.4.0: This setting is now ignored. Use *FINALIZE_METADATA* and *FINALIZE_METADATA_CONFIG* instead.

NORMALIZE_INPUT_FILE_MASK
Deprecated since version 0.4.0: This setting is now ignored. Use *FINALIZE_METADATA* and *FINALIZE_METADATA_CONFIG* instead.

PLUGIN_PERMISSIONS
Default: `{ 'MODIFY_RAW_POST': [] }`

This dictionary setting controls what permissions a plugin has. As of Engineer 0.5.0, the only key value is `MODIFY_RAW_POST`. If a plugin requires writing back to a post source file, you must explicitly list it in this setting. Otherwise the plugin will fail to update post source files.

Each permission value can optionally contain a `*` as a wildcard. This means that all plugins will automatically be granted that permission.

See also:

Plugin Permissions

New in version 0.5.0.

PUBLISH_DRAFTS
Default: `False`

If `True`, posts that have draft *status* will be considered published during site generation. This can be useful to test out how a new post might look on the site without worrying that you'll forget to change its status back to draft before you do a *real* build.

PUBLISH_PENDING
Default: `False`

Ordinarily Engineer only generates output for posts whose *timestamp* is in the past. Published posts that have a future date are considered 'pending.' When `PUBLISH_PENDING` is `True`, Engineer will output these future posts regardless of the current time.

PUBLISH_REVIEW
Default: `False`

If `True`, posts marked with a status of *review* will be output. This is useful for draft posts that you want to preview in the context of a site. These posts can be marked `review` and a settings file with `PUBLISH_REVIEW` set to true can be used to output them for review purposes. Using *review* instead of *published* and *PUBLISH_PENDING* helps preview posts without setting arbitrary dates in the future and eliminates concerns about accidental publication.

POST_TIMEZONE
Default: System default timezone

If your posts are primarily posted from a specific timezone, you can set this setting to instruct Engineer to assume that the timestamps in posts are in this timezone.

See also:

A Note About Timezones

SERVER_TIMEZONE
Default: POST_TIMEZONE

If the server hosting your site is in a different timezone than you are, you can set this setting so Engineer knows to adjust times appropriately. This is necessary mostly for *Emma*; it shouldn't affect generating your site in most cases.

See also:

A Note About Timezones

TIME_FORMAT

Default: `%I:%M %p %A, %B %d, %Y %Z`

TODO

PLUGINS

Default: None

A list of modules that contain Engineer *Plugins*.

See also:

Plugins

Settings File Inheritance

A settings file can inherit settings from another file. The inheritance model is what one would expect - it is similar to class inheritance in most programming languages.

In order to do this, you set the `SUPER` setting in your settings file to the path of the settings file. For example, you might have a file called `base.yaml` that contains your *SITE_TITLE*, *POST_DIR*, *SITE_URL*, *HOME_URL*, etc., and a second file called `production.yaml` that looks like this:

```
SUPER: base.yaml
OUTPUT_DIR: <path to output dir>
```

When you do a build using `production.yaml`, the settings from `base.yaml` will be loaded first, then the settings from `production.yaml` will be loaded. The settings in `production.yaml` will always 'win,' so any settings present in both will use the value specified in `production.yaml`.

Inheritance can span more than two files. In our example, if `base.yaml` inherited from another settings file, those settings would be loaded, then `base.yaml`, *then* `production.yaml`. This nesting can be arbitrarily deep, though it gets unwieldy after about three or four levels.

A given settings file can only directly inherit from a single parent.

Posts

Posts are the bread and butter of an Engineer site. Posts are Markdown files with either a `.md` or `.markdown` file extension and are structured like this:

```
title: What's Next?
timestamp: 09:41 AM Friday, March 09, 2012 PST
status: published
tags:
- example
- sample
slug: what-s-next


---

So what should you do now that you have created a sample Engineer site?
```

Posts are typically stored in a folder called `posts` within your site's source directory, but you can put them anywhere - even in multiple folders - by changing the *POST_DIR* setting.

Everything above the `---` is *Metadata*, and everything below it is the post itself (in Markdown, of course).

Metadata

Posts should contain some metadata that tells Engineer about the post. This metadata must be in YAML format, and must be the first thing in your post file. The YAML document separator (`---`) separates the post content and metadata.

None of the metadata is strictly *required* since there are defaults for everything but you must have at least one piece of metadata in your post file. In addition, Engineer will automatically update the metadata in your source post file during a build. This behavior is customizable; see the *Metadata Finalization* plugin for more information.

Changed in version 0.3.0: The metadata can now have a YAML document separator (`---`) above it as well as below it. This format is used by Jekyll, and by extension, Octopress, so posts written for those systems will migrate to Engineer without problems.

See also:

Compatibility With Other Static Site Generators

Metadata Parameters

`title` The title of the post. If you don't provide this, engineer will try to generate one based on the name of your post file, replacing any dashes or underscores in the file name with spaces. For example, if your post file is named `A-Day-In-the-Life.md`, Engineer will set the post title to 'A Day In the Life' unless you explicitly define a `title` in the post metadata.

`timestamp` The date and time that the post is/was published. The format of the date and time has pretty loose requirements, but in general it's best to follow this basic format: `2012-03-21 13:43:04`. The timestamp can be a future time, in which case the post will not become published until that time. If you don't provide an explicit timestamp, Engineer will set it to the date and time that the site is next built.

Note: Unless you specifically provide a timezone offset in your `timestamp` value, the time will be assumed to be in the same time zone as your *POST_TIMEZONE* setting.

See also:

A Note About Timezones

`status` The status of the post. Valid values are:

`draft` Draft posts are never output when a site is built. Status always defaults to *draft* if it's missing or set to an unknown value to avoid accidentally publishing something that wasn't meant to be published.

`published` Published posts are always output when a site is built *unless* they have a *timestamp* in the future, in which case they are not output. This behavior can be customized using the *PUBLISH_PENDING* setting.

`review` Posts marked `review` are only output if the setting *PUBLISH_REVIEW* is set to true.

New in version 0.3.0.

See also:

PUBLISH_DRAFTS, *PUBLISH_PENDING*, *PUBLISH_REVIEW*

slug You can set an explicit slug for the post. The slug represents the URL for your post and thus should only contain URL-safe characters. If not set, Engineer will generate a slug for you based on the name of your page. In general the only time you'll need or want to consider manually setting this is if you have multiple posts with the same name (and published on the same day), which Engineer cannot currently handle on its own.

tags A list of tags to be applied to the post. Completely optional. Tags will be used to generate tag pages - pages with all of the posts tagged with a specific tag listed. You can specify a single tag or multiple tags. If you specify multiple tags, they must be in YAML list format.

link If the post has an associated external link, the main post title on the site and in the RSS feed will link to the external link instead of the permalink to the post on your site. This method of linking was popularized by John Gruber on http://daringfireball.net.

via If you're including an external link, you might also want to provide some attribution to the person or site that helped you find the link. In this case, you can provide the `via` metadata property. It should be a string - the name of the person or site that you want to credit.

via-link If you want to link to a different URL as part of your attribution, you can provide an optional link to the blog or individual's personal site (or perhaps the article that linked you to the external link originally). Exactly how this attribution metadata is used in the site depends on the theme.

Changed in version 0.3.0: Prior to version 0.3.0, this property was `via_link`. Both forms of the property are supported in version 0.3.0+.

Custom Metadata

In addition to the metadata properties listed above, each post can include other custom metadata, specified in YAML just like regular metadata. Engineer will add these custom properties to the Post's *`custom_properties`* property, where they can be used by themes or plugins.

Custom properties are not manipulated in any way by Engineer itself (though plugins may change/update them) and they are maintained during *Metadata Finalization*.

A Note About Timezones

Time zones are a tricky thing in the best of circumstances, and unfortunately one of Python's few weaknesses is how it deals with them. It's particularly difficult to get the current system time zone, especially on Windows, so Engineer forces you to set a time zone explicitly. If you don't, Engineer assumes that times are in UTC. You can use the *`POST_TIMEZONE`* setting to set which timezone Engineer should assume your post timestamps are in.

You can see a complete list of the valid timezone settings at the PostgreSQL site. Yes, it's a bit weird, but the list there is the most comprehensive one I've seen that doesn't threaten to utterly confuse and overwhelm mere mortals when they see it. Keep in mind that some rows in the table list multiple valid strings that happen to correspond to the same time zone. For example, `Asia/Jerusalem`, `Asia/Tel_Aviv`, and `Israel` all correspond to the same timezone, and all are valid strings. (Why they could not simply put a delimiter *other than a space* between the strings in a single row I'll never understand.)

You can also choose to put a date/time string *with a UTC offset* in relevant places, in which case Engineer will understand that the time is in a specific zone. For example, if you specify your post's *timestamp* as something like `2012-04-17 08:47:00-08:00`, Engineer will understand that the time specified is 8 hours behind UTC. Generally I have found this to be a hassle, since forgetting the offset can cause incorrect post timestamps, and setting the *`POST_TIMEZONE`* is much more straightforward.

You might also find yourself in a situation where you write your posts in one timezone, but your server is in another. This generally isn't a problem *unless* you're using *Emma*. In that case you should be sure to set your *`SERVER_TIMEZONE`* as well.

Post Content

Your post content should be written in Markdown, and all the usual Markdown syntax rules apply. Engineer does provide some helpful CSS styles and template tags that might be useful when writing your posts. In addition to these, individual themes might provide their own.

Including Images

If you have images in your posts, you may find Engineer's built-in `img` function useful. This function allows you to easily output consistent image markup in your posts. You must have the *Jinja Post Processor Plugin* enabled in order to use this function.

Theme Designer Note

The exact HTML content output by the `img` function is customizable by a theme. See the *Theme creation documentation* for more details.

Syntax

The typical syntax for the `img` function looks like this:

```
{{ img(path_to_image,
      classes=[list of CSS classes to apply],
      width,
      height,
      title,
      alt,
      link) }}
```

All of the arguments except the path are optional. You can also pass the image source in as a string by treating the `img` function as a Jinja filter. In this case, the syntax would look like this:

```
{{ path_to_image|img(classes=[list of CSS classes to apply],
                     width,
                     height,
                     title,
                     alt,
                     link) }}
```

Examples

Consider an `img` tag like this:

```
{{ "http://farm8.staticflickr.com/7241/7206331966_c6419e544e.jpg"|img('center', 500,␣
↪375,
    'An xkcd.com comic in Reeder', 'An xkcd.com comic in Reeder',
    link='http://www.flickr.com/photos/76037594@N06/7206331966/') }}
```

Using the Dark Rainbow theme, this would output the following HTML:

```
<div class="image caption center">
    <a href="http://www.flickr.com/photos/76037594@N06/7206331966/">
        <img src="http://farm8.staticflickr.com/7241/7206331966_66d2e5e577.jpg"
             width="500"
             height="375"
             alt="An xkcd.com comic in Reeder">
    </a>
    <p>An xkcd.com comic in Reeder</p>
</div>
```

Teaser Content

Some themes support 'teaser content.' As part of a post, you can specify a break in the content. Only content before the break will be displayed on list pages, such as the homepage, but individual post pages will contain the full post.

You can specify the break in your post with either `-- more --` or the Octopress-style `<!-- more -->`. See *Post Breaks/Excerpts/Teasers* for more information.

New in version 0.3.0.

Themes

Engineer currently includes two bundled themes: *Dark Rainbow* and *Ole B*. You can also *create your own themes* if you like.

See also:

Creating Your Own Themes

Bundled Themes

Dark Rainbow

The default Engineer theme, **Dark Rainbow** has also been called 'Voldemort's Skittles,' 'Unicorn Vomit,' and other names not fit to repeat here. Needless to say, the parade of colors isn't for everyone. That said, Dark Rainbow showcases several of the key features Engineer provides, including customizable navigation with contextual highlighting, LESS support, TypeKit and JQuery integration, and Foundation CSS-based layouts.

Dark Rainbow was created by Tyler Butler and is available under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

Note: By default *Dark Rainbow* uses several fonts available at TypeKit. These fonts are available as part of TypeKit's trial plan.

Settings

Dark Rainbow supports the following settings which can be configured using the *`THEME_SETTINGS`* setting.

`comments` ***(string)*** Set this setting to either `disqus` or `intensedebate` to enable comments on your site. Comments are off by default. Disqus or Instense Debate are supported comment systems. Be sure to also set the `comments_account` setting properly as well.

comments_account *(string)* Both Disqus and Intense Debate require an account ID in order to associate comments properly with your site. Set this setting to the account ID for your respective comment account.

simple_search *(bool)* A boolean indicating whether simple search should be enabled for the site. *Defaults to True.*

Note that if you have customized your sidebar, you must include the `_search.html` snippet in your sidebar file or the search box will not be visible. See *Snippets* for more information.

New in version 0.4.0.

sharing A group of settings that control whether Facebook/Twitter share buttons are displayed on posts. `enabled` *(bool)*

Turns the sharing buttons on or off completely. *Defaults to False.*

facebook

enabled *(bool)* Turns the Facebook sharing button on or off. *Defaults to False.*

app_id *(string)* In order to share on Facebook, you must have a developer app ID specified here.

twitter

enabled *(bool)* Turns the Twitter sharing button on or off. *Defaults to False.*

username *(string)* If supplied, the Twitter share dialog will prepopulate your username in the tweet suggestion.

New in version 0.6.0.

typekit_id *(string)* The ID of the TypeKit kit that should be used. Dark Rainbow uses specific fonts that should be included in the kit.

twitter_id *(string)* The username of the Twitter user whose feed should be shown in the sidebar. *Defaults to tylerbutler if not provided.*

Deprecated since version 0.5.0: This setting is obsolete and ignored. The Tweet library has been removed from Engineer. See the *Release Notes* for more information.

tweet_count *(int)* The number of tweets to include in the Twitter sidebar. *Defaults to 4 if not provided.*

Deprecated since version 0.5.0: This setting is obsolete and ignored. The Tweet library has been removed from Engineer. See the *Release Notes* for more information.

Fonts

Dark Rainbow requires the following fonts:

- Museo Slab
- Myriad Pro
- Kulturista Web
- Ubuntu Mono *(optional)*

Changed in version 0.4.0: The Anonymous monospace font has been replaced by Ubuntu Mono by default. If you are using TypeKit you'll need to update your kit to include the Ubuntu Mono font. If you wish to continue using Anonymous you'll need to add your own CSS stylesheet.

Templates

Required Templates and Fragments

No templates are strictly required for this theme beyond the base *Template Fragments* that all Engineer sites will likely want to provide. In particular, users of the Dark Rainbow theme will probably want to create `_sidebar.html` and `_primary_nav.html` templates.

See also:

Template Fragments, *Navigation*, *Sidebar*

Inheritable Templates

Dark Rainbow includes several base templates that sites can inherit from to create *Template Pages*.

`template_page_simple.html` A simple template page layout that includes the default site sidebar.

`template_page_no_sidebar.html` A simple template page layout that removes the default site sidebar, devoting the entire page to the template page content.

Template Fragments

Dark Rainbow does not support any additional *Template Fragments* beyond those available for all Engineer sites.

Snippets

Dark Rainbow provides some small snippets that can be included in the sidebar of your site. These snippets are designed to be used in the sidebar, so using them is as simple as including them in your site's *_sidebar.html* template fragment. In order to maintain maximum compatibility with themes that might not provide these same widgets, you should specify `ignore missing` on the `include` directive.

For example, the Engineer sample site includes these widgets like so:

```
<section>
    <p>Welcome to the Engineer sample site.</p>
    <hr/>
    <nav>
        <ul>
            <li><a href="{{ urlname('about') }}">about</a></li>
            <li><a href="{{ urlname('themes') }}">themes</a></li>
        </ul>
    </nav>
</section>

{% include 'snippets/_search.html' ignore missing %}
{% include 'snippets/_feed_links.html' ignore missing %}
```

The following snippets are available:

`snippets/_feed_links.html` Adds a link to your RSS feed.

`snippets/_search.html` Adds a search box to your site sidebar.

Manifest

```
name: 'Dark Rainbow'
id: 'dark_rainbow'
description: 'A dark theme with just a hint of color.'
author: 'Tyler Butler <tyler@tylerbutler.com>'
website: 'http://tylerbutler.com'
license: 'Creative Commons BY-SA 3.0'
use_precompiled_styles: no

template_dirs:
  - '../_shared/templates/'

copy_content:
  - ['../_shared/images/rss/37.png', 'images/rss.png']

settings:
  typekit_id: ~
  comments: ~
  comments_account: ~
  simple_search: yes
  bigfoot:
    enabled: yes
  sharing:
    enabled: no
    facebook:
      enabled: no
      app_id: ~
    twitter:
      enabled: no
      username: ~
```

Ole B

Ole B is a bright, simple theme based on Ole Begemann's design for http://oleb.net, created with his permission. The theme was written from scratch by Tyler Butler using Ole's site as a reference.

Note: By default *Ole B* uses several fonts available at TypeKit. These fonts are available as part of TypeKit's trial plan.

Settings

Ole B supports the following settings which can be configured using the *THEME_SETTINGS* setting.

comments ***(string)*** Set this setting to either `disqus` or `intensedebate` to enable comments on your site. Comments are off by default. Disqus or Instense Debate are supported comment systems. Be sure to also set the `comments_account` setting properly as well.

comments_account ***(string)*** Both Disqus and Intense Debate require an account ID in order to associate comments properly with your site. Set this setting to the account ID for your respective comment account.

simple_search ***(bool)*** A boolean indicating whether simple search should be enabled for the site. *Defaults to True.*

Note that if you have customized your sidebar, you must include the `_search.html` snippet in your sidebar file or the search box will not be visible. See *Snippets* for more information.

New in version 0.4.0.

`typekit_id` ***(string)*** The ID of the TypeKit kit that should be used. Ole B uses specific fonts that should be included in the kit.

`twitter_id` ***(string)*** The username of the Twitter user whose feed should be shown in the sidebar. *Defaults to tylerbutler if not provided.*

Deprecated since version 0.5.0: This setting is obsolete and ignored. The Tweet library has been removed from Engineer. See the *Release Notes* for more information.

`tweet_count` ***(int)*** The number of tweets to include in the Twitter sidebar. *Defaults to 4 if not provided.*

Deprecated since version 0.5.0: This setting is obsolete and ignored. The Tweet library has been removed from Engineer. See the *Release Notes* for more information.

Fonts

Ole B requires the following fonts:

- Museo Slab
- Museo Sans

Templates

Required Templates and Fragments

No templates are strictly required for this theme beyond the base *Template Fragments* that all Engineer sites will likely want to provide. In particular, users of the Ole B theme will probably want to create `_sidebar.html` and `_primary_nav.html` templates.

See also:

Template Fragments, *Navigation*, *Sidebar*

Inheritable Templates

Ole B includes several base templates that sites can inherit from to create *Template Pages*.

`template_page_simple` A simple template page layout that includes the default site sidebar.

Template Fragments

Ole B does not support any additional *Template Fragments* beyond those available for all Engineer sites.

Snippets

OleB provides some small snippets that can be included in the sidebar of your site. These snippets are designed to be used in the sidebar, so using them is as simple as including them in your site's *_sidebar.html* template fragment. In

order to maintain maximum compatibility with themes that might not provide these same widgets, you should specify `ignore missing` on the `include` directive.

For example, the Engineer sample site includes these widgets like so:

```
<section>
    <p>Welcome to the Engineer sample site.</p>
    <hr/>
    <nav>
        <ul>
            <li><a href="{{ urlname('about') }}">about</a></li>
            <li><a href="{{ urlname('themes') }}">themes</a></li>
        </ul>
    </nav>
</section>

{% include 'snippets/_search.html' ignore missing %}
{% include 'snippets/_feed_links.html' ignore missing %}
```

The following snippets are available:

`snippets/_feed_links.html` Adds a link to your RSS feed.

`snippets/_search.html` Adds a search box to your site sidebar.

Manifest

```
name: 'Ole Begemann'
id: oleb
description: "A bright design based on Ole Begemann's oleb.net. Used with permission."
author: 'Tyler Butler <tyler@tylerbutler.com>'
website: 'http://tylerbutler.com'
license: 'N/A'
use_precompiled_styles: no

template_dirs:
  - '../_shared/templates/'

copy_content:
  - ['../_shared/images/rss/04.png', 'images/rss.png']

settings:
  typekit_id: ~
  comments: ~
  comments_account: ~
  simple_search: yes
  bigfoot:
    enabled: no
```

Using Themes

By default Engineer uses the *Dark Rainbow* theme. Changing the theme to something else is as simple as changing the *THEME* setting in your *settings file*.

Most themes do not require any customization, though they might provide *Templates* that you might find useful. For example, the *Dark Rainbow* theme provides a few different layouts for template pages that you can use as a basis for your template pages.

Installing New Themes

Engineer themes can be used without installation. Simply download the theme, place it in the `themes` directory within your site directory, and change your *THEME* setting to use the new theme.

Alternatively, some themes might be available as an installable *plugin*. If this is the case for the theme you want to use, then follow the installation instructions for the theme. Once installed, it will be available to any Engineer site.

Templates

Engineer makes heavy use of Jinja2 templates to render a site. Most templates come as part of *Themes*, and you might not even need to worry about them. In fact, Engineer makes it easy to customize your site without creating full-blown Jinja templates.

Note: Doing anything somewhat advanced with templates will require some knowledge of the Jinja2 template syntax and features. The Jinja2 Documentation is an excellent place to learn more about the language. The Template Designer Documentation in particular is a useful starting point if you're ready to jump right in. As usual, you can look at the *sample site* as a reference point to see how things fit together.

Template Fragments

Template fragments are blocks of HTML that you want to put into pages on your site. For example, the `_footer.html` fragment contains markup you want to appear in the footer of your site. While template fragments are complete Jinja2 templates, and thus can contain any Jinja2 syntax, you don't have to. In fact, with the exception of *Navigation*, you can simply put raw HTML into your template fragments - Engineer will pull the content of those fragments into your site.

Built-in Template Fragments

Engineer makes several template fragments available to all sites. While individual themes might also expose their own, the following are available regardless of the theme you're using. In order to 'use' a fragment, all you need to do is create a file with the same name as the fragment you're using in your *TEMPLATE_DIR*. For example, creating a `_footer.html` file in your template directory will make Engineer put the contents of that file in the footer of your site.

Tip: Template fragments should always be put in your site's *TEMPLATE_DIR*. The built-in fragments should all be in the root of the template dir, but themes might also support other fragments that should be located in slightly different places. Check your theme's documentation for details.

By convention all template fragments' names start with an underscore (_) and are optional. That said, the `_sidebar.html` and `_nav_primary.html` fragments should be created. Otherwise you'll most likely see sample content for your site's sidebar and navigation.

Theme Designer Note

Template fragments are meant to contain only HTML (with the notable exception of *Navigation*). If you're including your own fragments, you should ensure users don't need to use special Jinja syntax or unique macros in their fragments if at all possible. If you do require such advanced syntax, be sure it's clearly documented.

_scripts_top.html Use this fragment to put additional scripts at the top of your pages. This can be useful for getting *web analytics* scripts into your site, for example.

_scripts_bottom.html This fragment is similar to `_scripts_top.html` except the scripts are included at the bottom of your pages rather than at the top.

_stylesheets.html Use this fragment to put additional CSS or LESS stylesheets at the top of your pages.

`_nav_primary.html`

_nav_primary_links.html These two fragments together contain the outer navigation links for your site. See the documentation on *Navigation* for more details on fragment and what they should contain.

_sidebar.html This fragment contains a sidebar for your site. See the documentation on *Sidebar* for more details on this fragment and what it should contain.

_footer.html This fragment contains the footer content for your site.

Note: The Engineer developers (just me, really) would really appreciate it if you linked to the Engineer project in your footer. If you're finding Engineer useful, then linking back to the project is a great way to spread the word. You can put a link in manually if you'd like, or you can simply paste the following snippet into your `_footer.html` fragment:

```
{% include 'snippets/_powered_by.html' %}
```

That will insert a little 'Powered by Engineer' link into your footer. Don't feel obligated to do this, of course, but if you do I really do appreciate it!

Navigation

Warning: Navigation is an area of active development in Engineer. The current system is kludgy at best and I plan to give it a proper overhaul in an upcoming Engineer release.

Navigation links are critical to any website. In Engineer, the primary navigation links for your site should be put in the `_nav_primary_links.html` template fragment. This file should contain a set of `<li>` elements, each of which is a navigation link. You can hard-code these links if you'd like, but Engineer includes some Jinja macros that make generating more dynamic navigation links possible.

If you need more control over your navigation, you can also override the contents of `_nav_primary.html` by providing your own. By default, `_nav_primary.html` merely contains some outer scaffolding for navigation links (i.e. a `<ul>` tag):

```
<ul class="replace_me">
    {% include "_nav_primary_links.html" with context %}
</ul>
```

You can of course replace this with whatever you wish, but keep in mind that some themes may expect certain CSS classes to be applied to the navigation.

Using `navigation_link`

Since Engineer sites are statically generated, creating dynamic navigation links with highlighting for current nodes is a bit challenging. The `navigation_link` macro makes this easier. A macro is a Jinja2 construct that is similar

to a function in a programming language. The `navigation_link` macro, when called, outputs a list item (`<li>` element) with a link.

It's a bit easier to see it in action. Here's what the sample site `_nav_primary_links.html` template fragment looks like:

```
{% from 'core/_macros.html' import navigation_link with context %}

{{ navigation_link('this is', urlname('home'), []) }}
{{ navigation_link('sample', urlname('home'), []) }}
{{ navigation_link('navigation', urlname('home'), []) }}
```

We first import the `navigation_link` macro from `core/_macros.html`, then subsequently call the macro to create the individual list items in the navigation list. When this fragment is rendered on the homepage of the site, the HTML looks like this:

```
<li class="current"><a href="/">articles</a></li>
<li><a href="/about">about</a></li>
<li><a href="/themes">themes</a></li>
```

The `navigation_link` macro takes four arguments: the text to display for the link, the actual URL of the link, a list of contexts in which the link should be highlighted, and an optional CSS class name that should be applied to the active navigation nodes. By default, a highlighted link simply has the CSS class specified in the *`ACTIVE_NAV_CLASS`* setting applied to it; this can be overridden with each call to `navigation_link`.

Navigation Contexts

The way that Engineer determines whether a link should be highlighted or not is based on the current navigation context. Whenever Engineer is rendering a page it has a context. If that context is in the list of contexts passed to `navigation_link`, then Engineer highlights that link. Thus, in the example above, the *articles* link should be highlighted whenever the current navigation context is `post` or `listpage`.

Available navigation contexts:

`post` This context is active whenever Engineer is rendering a post.

`listpage` This context is active whenever Engineer is rendering a list of posts. For example, the home page of the Engineer site will have this navigation context.

`archive` This context is active whenever Engineer is rendering the archives page.

`tag` This context is active whenever Engineer is rendering a tag page.

template page name In addition, all template pages are rendered with a navigation context matching their name. In the sample site, this is used to highlight the *about* and *themes* navigation links when you're visiting those template pages in the site.

The `urlname` Function

The `urlname` function provides a quick way to get a URL for a given page in your site. It is especially handy for navigation. The acceptable arguments are:

`'home'` URL to the home page of the site.

`'archives'` URL to the archives page.

`'feed'` URL to the site RSS feed.

'listpage' URL to a specific slice of the home page. Since Engineer paginates the home page, this argument allows one to create a link directly to a specific page in the pagination. The slice number is provided as a second argument. For example:

```
urlname('listpage', 2)
```

'tag' URL to the tag page for the given tag. The tag name is provided as a second argument. For example:

```
urlname('tag', 'engineer')
```

Sidebar

The `_sidebar.html` should contain HTML markup you wish to display in a sidebar on your site. This content should be wrapped in a `<section>` container as appropriate. For example, the sample site `_sidebar.html` looks like this:

```
<section>
    <p>Welcome to the Engineer sample site.</p>
    <hr/>
    <nav>
        <ul>
            <li><a href="{{ urlname('about') }}">about</a></li>
            <li><a href="{{ urlname('themes') }}">themes</a></li>
        </ul>
    </nav>
</section>

{% include 'snippets/_search.html' ignore missing %}
{% include 'snippets/_feed_links.html' ignore missing %}
```

Sitemap Templates

If you need to customize the sitemap that Engineer generates for you, you can provide your own templates that Engineer will use to generate it. This template should be named `sitemap.xml` and should be in the root of your site's *TEMPLATE_DIR*.

New in version 0.3.0.

Snippets

In addition to *Template Fragments*, some themes might provide 'snippets': small pieces of content or layout that you might want to include in your sidebar, footer, etc. For example, the *Dark Rainbow* theme provides snippets for a search bar and RSS feed links to include in your sidebar. Also, the 'Powered by Engineer' footer is a snippet. By convention, snippets are placed in the 'snippets' folder. Because some themes might not provide snippets, you should use the `ignore missing` command when including them in your site. For example:

```
<section>
    <p>Welcome to the Engineer sample site.</p>
    <hr/>
    <nav>
        <ul>
            <li><a href="{{ urlname('about') }}">about</a></li>
            <li><a href="{{ urlname('themes') }}">themes</a></li>
        </ul>
```

```
    </nav>
</section>

{% include 'snippets/_search.html' ignore missing %}
{% include 'snippets/_feed_links.html' ignore missing %}
```

New in version 0.4.0.

Template Pages

Many sites have a need for 'flat' pages like an 'about' or 'contact us' page. The 'flat' terminology isn't quite right in Engineer's case, since all pages in Engineer are flat, but the need is real. Engineer provides this capability via template pages.

A template page is basically just a simple HTML page in your site, but unlike a standard HTML page, you can use Jinja2 templates to inherit the look and feel of your site but add content specific to your page. As usual, it's easier to look at an example. Here's the `themes.html` template page from the Engineer sample site:

```
{% extends 'theme/template_page_simple.html' %}

{% block page_title %}Themes{% endblock %}

{% block header_secondary_title %}Themes{% endblock %}

{% block content %}
    <article>
        <p>Engineer comes with two themes, and provides a basic framework for creating
        additional ones if you're so inclined.</p>

        <h2>Dark Rainbow</h2>

        <p>The default Engineer theme, Dark Rainbow has also been called 'Voldemort's␣
↪Skittles,'
            'Unicorn Vomit,' and other names not fit to repeat here. Needless to say,␣
↪the parade of
            colors isn't for everyone.</p>
    </article>
{% endblock %}
```

As you can see, this page extends `theme/template_page_simple.html`, which is one of the inheritable templates included with the *Dark Rainbow* theme. It sets the page title to 'Themes' and adds some basic content for the page in the `content` block.

All themes include a basic template page base called `template_page_base.html` that exposes the following blocks:

`page_title` The title of the page.

`content` The content of the page.

Themes may expose their own additional template page bases, like *Dark Rainbow* does, but at the very least `template_page_base.html` will always be available.

Template pages should be placed in your *`TEMPLATE_PAGE_DIR`*. Folders are permitted, so you can organize your template pages and that structure will be reflected in the URL paths to your pages.

Tip: If you'd like to write content for template pages in Markdown, you can. Simply wrap your Markdown content

with the `markdown` filter. For example:

```
{% filter markdown %}
    This site is built using [Engineer](/projects/engineer), a static site generator␣
↪I wrote myself after
    being inspired by [Brent Simmons][], Marco Arment's Second Crack, Jekyll,␣
↪Octopress,
    and Hyde. It's written in [Python][] and uses [Jinja2][] for templating. I use␣
↪the management site
    available with Engineer (aka Emma) to manage my posts, which in turn runs on␣
↪[Bottle][].
{%endfilter %}
```

Included Plugins

Engineer includes a few optional plugins you can use to further customize its behavior. If you have an idea for your own plugin, you might consider *creating it yourself*.

Note: Some plugin capabilities require you to explicitly give the plugin *special permissions*. Check the plugin's documentation to see if this is the case. The plugin permissions system is new to Engineer 0.5.0.

Metadata Finalization

Engineer will automatically fill in pieces of metadata about your posts during the build process, and this plugin can also 'finalize' some of that metadata and write it back to your post file.

For example, if you have a post that you just wrote and are ready to publish, you likely want Engineer to use the current date/time as the timestamp for the post. However, in order to ensure future build processes know the right publish time for that post, the metadata needs to be in the file itself, so Engineer will automatically add `timestamp: <current date/time>` to your post file when it builds the site.

On the other hand, you might have a post that is in review or draft form, and you're building the site to preview it. In that case, you *don't* want the timestamp to be added to the file.

Settings

The finalization process is customizable. The *FINALIZE_METADATA_CONFIG* setting defines which metadata settings are finalized and what posts (based on their status) the finalization process applies to. For example, the default `FINALIZE_METADATA_CONFIG` looks like this:

```
FINALIZE_METADATA_CONFIG:
  timestamp:
    - published
  title:
    - published
    - review
    - draft
  slug:
    - published
    - review
    - draft
```

```
  url:
    - published
    - review
```

This metadata map tells Engineer to finalize timestamps *only* for published posts and normalizes titles and slugs for all posts. You can override this default map by providing your own map in your own *settings file*. In addition, you can turn metadata normalization on and off completely using the `FINALIZE_METADATA` setting.

Note: Metadata that already exists in post files will always be maintained regardless of this setting. For example, if you are using the default settings but have a draft post that already has a `url` value, that metadata will be maintained in the output file, even though URLs will not be set for drafts in general.

---

class `engineer.conf.`**`EngineerConfiguration`**

`FINALIZE_METADATA`
Default: `True`

Turns *Metadata Finalization* on and off.

See also:

Metadata Finalization, `FINALIZE_METADATA_CONFIG`

`FINALIZE_METADATA_CONFIG`
Default:

```
FINALIZE_METADATA_CONFIG:
  timestamp:
    - published
  title:
    - published
    - review
    - draft
  slug:
    - published
    - review
    - draft
  url:
    - published
    - review
```

A mapping of post metadata values to the post statuses in which they'll be finalized. By default, Engineer will finalize timestamps *only* for published posts and normalizes titles and slugs for all posts.

See also:

Metadata Finalization, `FINALIZE_METADATA`

`METADATA_FORMAT`
Default: `'input'`

Specifies which metadata format to output. As of version 0.5.0, this only controls whether or not to force *Post Metadata 'Fencing'*. When set to the default, `'input'`, the finalized metadata format will match that of the input.

Other valid values for this setting are:

`'fenced'` Always output the metadata as fenced.

'unfenced' Always output the metadata as unfenced.

New in version 0.4.0: In version 0.4.0, the old post normalization process has been superceded by the *Metadata Finalization* and *Post Renamer* plugins.

Changed in version 0.5.0: Added the *METADATA_FORMAT* setting.

Post Breaks/Excerpts/Teasers

If you wish to show only an excerpt of a post on a rollup page, you can insert a break marker into your post content and Engineer will break it up for you.

Engineer supports Octopress-style `<!--more-->` post breaks in addition to the simpler `-- more --`. Use whichever one you wish. Only the first section of the post, before the 'more' break marker, will be displayed on a rollup page.

By default the RSS feed that Engineer generates will only include teaser content. However, you can override this and make your feed full content by setting the `FEED_FULL_CONTENT` setting to true in your Engineer settings file.

The Post Breaks plugin does not need to be activated in any way; it always runs but has no effect on posts that don't include a break marker.

New in version 0.3.0.

Changed in version 0.4.0: Added the `FEED_FULL_CONTENT` setting.

See also:

Compatibility With Other Static Site Generators

Post Renamer

It can be handy when your post source files have names that tell you a little about the post itself. While you can obviously name post files whatever you like, Engineer can automatically rename your files during the build process to help keep things organized. When combined with *Metadata Finalization*, Engineer can do a lot of heavy lifting to keep your posts organized and easy to manage.

The Post Renamer plugin is disabled by default, and can be enabled by setting the `POST_RENAME_ENABLED` setting to true. When enabled, the plugin uses the `POST_RENAME_CONFIG` setting to determine how to rename files. This configuration setting is similar in form to the *PERMALINK_STYLE* setting, and specifies a mapping of *post status* to a rename format string.

For example, the default `POST_RENAME_CONFIG` setting is:

```
POST_RENAME_CONFIG:
  draft: '({status}) {slug}.md'
  review: '({status}) {year}-{month}-{day} {slug}.md'
  published: '({status_short}) {year}-{month}-{day} {slug}.md'
```

With this configuration, a draft post with the title "Welcome to Engineer" would be renamed to `(draft) welcome-to-engineer.md`. The format strings should follow standard Python string formatting rules. The following named parameters are available for you to use in your format string:

year The year portion of the post's timestamp as an integer.

month The month portion of the post's timestamp as string - includes a leading zero if needed.

day The day portion of the post's timestamp as a string - includes a leading zero if needed.

i_month The month portion of the post's timestamp as an integer.

i_day The day portion of the post's timestamp as an integer.

slug The post's slug.

status The post's status as a string (e.g. `draft`).

status_short The post's status in a short form (e.g. `d` for `draft`, `p` for `published`, etc.).

timestamp The post's timestamp as a datetime.

post The post object itself.

If you wish for posts of a certain status to not be renamed at all, simply use a ~ (tilde - YAML's equivalent to `None` or null) in your `POST_RENAME_CONFIG` setting. For example, the following setting will not rename draft and review posts, but will rename published posts according to the default configuration:

```
POST_RENAME_CONFIG:
  draft: ~
  review: ~
```

New in version 0.4.0: In version 0.4.0, the old post normalization process has been superceded by the *Metadata Finalization* and *Post Renamer* plugins.

Changed in version 0.4.2: The plugin is now disabled by default. Renaming post files caused confusion and headaches for new Engineer users.

Global/Shared Links

If you find yourself often inserting the same links in your posts, you might benefit from using the Global Links plugin. Using this plugin, you can create a list of common links and store them in a file along with your site settings. You can reference these links in any post; they are always available to all posts.

New in version 0.5.0.

Usage

Activating the Plugin

In order to use global links, you first need to do two things:

1. Create a file to store the links. This file can be anywhere, but it is generally easiest to put it alongside your site's *settings file(s)*. The convention is to call the file `global_links.md` but this is by no means required.
2. Set the `GLOBAL_LINKS_FILE` setting in your settings file. It should be set to the path of your global links file. It can either be an absolute path, or a path *relative to the location of the settings file.*

Once you have done this, the links in the file you created in step 1 will be available to all posts.

Adding Links

The Global Links plugin utilizes a feature in Markdown called 'reference-style links'. Reference-style links look like this:

```
This is an example of a [reference-style link][rsl]. You can also use implicit link␣
↪names like [Google][] if you
prefer.
```

```
[rsl]: http://daringfireball.net/projects/markdown/syntax#link
[Google]: http://www.google.com
```

As you can see, reference-style links allow you to link to things using definitions defined later in the Markdown document. The Global Links plugin simply takes advantage of this. If you put the following in your `GLOBAL_LINKS_FILE`, it will be available to all posts:

```
[rsl]: http://daringfireball.net/projects/markdown/syntax#link
[Google]: http://www.google.com
```

Then you can write your posts and reference the links defined in your `GLOBAL_LINKS_FILE`.

Tip: The Global Links plugin isn't limited to just links. You can also put abbreviations or even footnotes (though I can't think of any reason why you'd want to use 'global footnotes'), in your `GLOBAL_LINKS_FILE`, and it will be available to your posts.

Markdown Lazy Links

This plugin allows you to use 'lazy links' in your posts. The idea comes from Brett Terpstra, and more detail is available at http://brettterpstra.com/2013/10/19/lazy-markdown-reference-links/. Unlike Brett's sample implementation, the Engineer plugin supports adding lazy links to posts that already have numeric reference links.

New in version 0.5.0.

Usage

The lazy links plugin is enabled by default, so you can start using them without any configuration changes.

By default, the lazy links are handled each time a build is run. In other words, the lazy links are transformed into numeric reference-style links each time; the links stay lazy in the original source post. If you wish to transform the lazy links into real numeric reference-style links in the source post files as part of a build, you'll need to tweak a few settings:

1. Ensure the *`FINALIZE_METADATA`* setting is enabled.
2. Set the `LAZY_LINKS_PERSIST` setting to `True` in your configuration file.
3. Give `engineer.plugins.bundled.LazyMarkdownLinksPlugin` the `MODIFY_RAW_POST` permission.

   See also:

   Plugin Permissions

A sample Engineer configuration file might look like this:

```
FINALIZE_METADATA: yes
LAZY_LINKS_PERSIST: yes

PLUGIN_PERMISSIONS:
  MODIFY_RAW_POST:
  - engineer.plugins.bundled.LazyMarkdownLinksPlugin
```

Jinja Post Processor Plugin

This plugin runs your post content through the Jinja template engine prior to tranforming it into HTML. This allows you to use Jinja filters, variables, and other content in your posts. For example, this plugin lets you use the handy *img* Jinja filter to insert images into your posts consistently.

New in version 0.5.0.

Usage

The Jinja Post Processor plugin is enabled by default. If you wish to disable it, you can set the `JINJA_POSTPROCESSOR_ENABLED` setting to `False` in your configuration file. Keep in mind that disabling the plugin will cause some built-in Engineer features such as the *img filter* to not work.

Engineer Commandline

Engineer is primarily invoked at the command line. The command is aptly called `engineer`, or `engineer.exe` on Windows. It accepts five basic top-level commands: `build`, `clean`, `serve`, `emma` and `init`, which each accept additional parameters.

Common Arguments

All of the Engineer commands accept the following arguments:

`-h, --help`
Display help for the command.

`-v, --verbose`
Display verbose command line output. You can see *extremely* verbose output by specifying the option twice. For example:

```
engineer build -vv
```

Changed in version 0.2.3.

`-s, --settings, --config`
Specify the path to the settings file to use. Defaults to `config.yaml` if not provided.

Note: While the *engineer init* command does accept this argument, it does not use it in any way.

Sub-commands

`engineer init`

Initialize a directory with a basic structure for an Engineer site, optionally including sample content. Note that using the `init` command is *not* required to create an Engineer site; all it does is a create a general purpose folder structure, a *settings file*, and optionally some sample content.

Usage:

```
engineer init [-h] [-v] [-s CONFIG_FILE] [-m {azure}] [--sample] [--force]
```

-m, --mode
Initializes a site structure designed for deployment to a specific hosting service such as Azure. See *Deploying Engineer Sites* for more details. Valid options:

- `azure`: Initializes a site for deployment to Azure.

--sample
By default, the `init` command does not create sample content to provide a starting point for a new site. By passing this option, however, sample content will be created.

Changed in version 0.5.0: Replaced the `--no-sample` option with this, effectively reversing the default.

-f, --force
Forcefully initialize a folder as an engineer site even if the target folder is not empty. **Use with caution!**

engineer build

Build an Engineer site from an input settings file and other source files.

Usage:

```
engineer build [-h] [-v] [-s CONFIG_FILE] [-c]
```

-c, --clean
Clear all caches and the output directory prior to building. This parameter is equivalent to *engineer clean* but immediately runs a `build` after.

engineer clean

Clears all caches and the output directory. This can be useful if you're seeing strange errors such as changes not being picked up properly or you simply want to 'start fresh.'

Usage:

```
engineer clean [-h] [-v] [-s CONFIG_FILE] [-p PORT]
```

engineer serve

Starts the built-in Engineer development server. The dev server will serve up a site's output directory contents at http://localhost:8000. You can press `Ctrl-C` to stop the dev server when you're done with it. Note that `serve` does not build a site, so you should run *engineer build* before you run *engineer serve*. Also keep in mind that if you make changes to the site source, such as posts or whatnot, you'll need to manually rebuild the site in order for those changes to be reflected. Adding the capability to autodetect changes and rebuild the site as needed are planned but not yet implemented.

Note: It's not a good idea to use the dev server to serve your site in production. While it's probably capable of this since it uses bottle.py under the covers, it hasn't been tested or designed for that purpose. Besides, part of the benefit in using Engineer in the first place is that you can just copy the output to an existing production web server and go. Why take on additional overhead of running your own server if you don't need to?

Usage:

```
engineer serve [-h] [-v] [-s CONFIG_FILE] [-p PORT]
```

-p, --port
Specify the port the development server should run on. If not specified, the default is 8000.

New in version 0.2.3.

engineer emma

Documentation TBD.

Usage:

```
engineer emma [-h] [-v] [-s CONFIG_FILE] [-p PORT] [--prefix PREFIX] (-r | -g | -u)
```

Deploying Engineer Sites

One of the benefits to Engineer is that the resulting site can be uploaded to pretty much any web host. You can use FTP to upload the site content anywhere, or use rsync to sync your local site to a remote server.

You can also deploy to Azure and GitHub Pages. Documentation for these options is not yet complete. However, see http://www.tylerbutler.com/tag/engineer/ for some information about doing this in the meantime.

EMMA: Engineer Miniature Management Automaton

Coming soon...

Compatibility With Other Static Site Generators

Engineer contains some compatibility features designed to ease transitions from other static site generators such as Jekyll/Octopress, as well as support tools designed for those systems.

Jekyll/Octopress

New in version 0.3.0.

Post Metadata 'Fencing'

Jekyll requires that post metadata (or YAML front matter, in Jekyll terms) be 'fenced' within a YAML document separator, like so:

```
---
title: Post Title
tags:
- tag 1
- tag 2
---
```

Engineer, in contrast, does not require the metadata to be preceded by a `---`. However, Engineer will handle Jekyll-style metadata with no trouble, and will maintain your post format during *Metadata Finalization*.

If you want Engineer to *always* output your metadata with or without fencing, you can use the *METADATA_FORMAT* setting. Simply set it to `fenced` or `unfenced` and Engineer will always output the format you specify, regardless of the input format.

New in version 0.3.0.

Post Breaks

Engineer supports Octopress-style `<!--more-->` post breaks in addition to the simpler `-- more --` Engineer style using the bundled *Post Breaks plugin*.

New in version 0.3.0.

Frequently Asked Questions

How Do I...

...change my site theme?

You can change the theme for a site using the *THEME* setting in your *settings file*. You'll need to specify the *ID* for the theme.

See also:

Settings Files, *Bundled Themes*

...customize my site's navigation links?

See *Navigation* and templates.

...use a custom RSS feed URL, e.g. Feedburner?

See the *FEED_URL* setting.

...add a flat page, like an 'about' or 'contact' page?

If you have an already-generated HTML page that you just want to put in your site, the *Raw Content* feature might be what you're looking for. More likely, though, you'll want to take advantage of *Template Pages*, which provide a simple way to create flat pages while inheriting the look and feel of your site theme.

See also:

Template Pages, *Raw Content*

...add custom JavaScript or CSS?

If you need to load additional JavaScript or CSS in your site, you can use the `_scripts_top.html`, `_scripts_bottom.html`, and `_stylesheets.html` *Template Fragments*.

For example, to load the Google Analytics JavaScript on your pages, you might add a `_scripts_top.html` template fragment to your site's `templates` folder, then paste the Google Analytics `<script>` tag into that file. All pages in your site will then include the Google Analytics JavaScript. Similarly, you can use the `_stylesheets.html` template fragment to include additional CSS or LESS stylesheets.

See also:

Template Fragments

...hook up Google Analytics (or another analytics system)?

See *How do I add custom JavaScript or CSS?* which explains how to add custom JavaScript, including the Google Analytics JavaScript, to your site pages.

...add a favicon or robots.txt file?

The *Raw Content* feature of Engineer can handle this. For example, to add a `robots.txt` file to the root of your site, put the file in the *CONTENT_DIR* of your site (defaults to `content`).

See also:

Raw Content

...put my site at a non-root path on my domain, such as `http://www.example.com/blog/`?

You can set the *HOME_URL* setting in your settings file as needed.

Release Notes

version 0.6.0 -

- Support for Facebook and Twitter 'share' buttons in the *Dark Rainbow* theme.
- *Dark Rainbow* now uses Foundation 6 and includes a number of enhancements for smaller mobile screens.
- Use webassets for static content management in themes.
- Added devtools commands to pre-compile theme CSS output.
- New command plugin model.
- Removed dependency on the *times* module; replaced with *arrow*.
- Bigfoot.js support in Dark Rainbow and OleB.
- Laid the groundwork for alternate post input formats, such as Textile or reStructuredText.
- Per-post templates and content templates.
- Plugin settings model changes; more consistency.
- New 'stashed content' feature for PostProcessor plugins.

- Add PostLink plugin.
- Bundled code highlighting styles for use by themes.
- Support for precompiled styles removed. It didn't make sense with the addition of webassets.

version 0.5.1 - May 28, 2014

- Fixed issue with precompiled LESS files.
- Updated documentation.

version 0.5.0 - April 10, 2014

- The bundled Tweet library has been removed and all related settings have been deprecated. Twitter has discontinued its unauthenticated v1.0 API, so Tweet stopped working as of June 11, 2013. If you have suggestions for a replacement library or solution for Twitter integration please file an issue on Github.
- The default setting of *PERMALINK_STYLE* has changed to `pretty` from `fulldate`.
- Atom feeds are now generated in addition to RSS feeds. Atom feeds are now the default. As part of this change, customization of the RSS feed using a template is no longer supported.
- New plugin added to support *Markdown Lazy Links*. This plugin is enabled by default, so you can start using lazy links immediately.
- A new 'experimental' plugin permissions model is in place. More developer information is available at: *Plugin Permissions*.
- Engineer's Jinja2 environment can now be modified using plugins. See the *JinjaEnvironmentPlugin* documentation.
- Jinja syntax can now be used in post content thanks to the new *Jinja Post Processor Plugin*.
- A new tag is available for *inserting images into posts*. The output can be *customized by themes*.
- You can now have Engineer automatically format your post metadata to be fenced (Jekyll-style) or unfenced using the *METADATA_FORMAT* setting.
- User and environment variables are now expanded when they appear in Engineer settings. For example, you can now use `~/engineer/posts/` as your *POST_DIR*. Huzzah!
- The *POST_DIR* setting can now automatically find posts in all subdirectories within a given path. See the docs for more details.
- Themes can now include precompiled versions of LESS stylesheets which will be used by default. See *Static Content* for more details.
- As usual, this release contains a number of bug fixes and tweaks.

version 0.4.6 - February 19, 2014

- Update to a new version of setuptools' bootstrapper. This should ease installation pains for new users that have more recent versions of setuptools.

version 0.4.5 - October 2, 2013

- Update to a new version of typogrify-engineer. Due to changes in the original typogrify package as well as in pip installer behavior, Engineer was failing to install properly from PyPI for new users.

version 0.4.4 - June 23, 2013

- Addresses compatibility issue with more recent versions of html5lib (issue 63). A more comprehensive fix will come in a future version.

version 0.4.3 - December 10, 2012

- Fixes issue 42 which managed to sneak into 0.4.2, causing an exception to be thrown for some configurations.

version 0.4.2 - December 10, 2012

Note: Engineer no longer requires the `zope.cachedescriptors` and `compressinja` packages. You can uninstall these packages if you wish. If you're using pip, simply type:

```
pip uninstall zope.cachedescriptors compressinja
```

- The *Post Renamer* plugin is no longer on by default.

  Important: If you wish Engineer to behave the way it did previously, simply set the `POST_RENAME_ENABLED` setting to true in you Engineer settings file.

- Fixes issue #36 which caused cache corruption on Mac OS X Lion.
- Fixes issue #39 which prevented the debug server from working properly on non-Windows operating systems.

version 0.4.1 - December 4, 2012

- *Finalization plugin*: No longer writes files if their metadata has not changed. This should prevent a rather annoying behavior where post files would always be modified during a build regardless if they had changed or not. This broke sorting the post files by 'last modified time', among other things.
- Fixes issues with automatic version handling.

version 0.4.0 - November 28, 2012

- Added support for custom URL/permalink schemes with the *PERMALINK_STYLE* setting.

  Important: Note that while the default has not changed in this release, it will in 0.5.0, so if you wish to continue to use the current Engineer URL scheme, you should update your settings files now.

- Broad changes to post and metadata normalization. These features have been broken out into two separate plugins, the *Metadata Finalization* plugin and the *Post Renamer*. Accordingly, the settings *NORMALIZE_INPUT_FILES* and *NORMALIZE_INPUT_FILE_MASK* have been deprecated. See the documentation for the two new plugins for more details.
- The *Dark Rainbow* and *Ole B* themes can now support comments using either Disqus or Intense Debate.
- The *Dark Rainbow* and *Ole B* themes now support simple site search using Google.
- Added the *ACTIVE_NAV_CLASS* setting to enable users to change the class that is applied to active navigation nodes. This should make it easier to integrate with CSS frameworks that use a different class name.
- Theme creators can now more easily share content between several themes using the *copy_content* and *template dirs* theme manifest settings.
- The *post breaks plugin* now outputs only the teaser content into the site RSS feed by default. This behavior can be changed using the `FEED_FULL_CONTENT` setting.
- Added a new `CommandPlugin` class. This enables other developers to write plugins that add new command line commands to Engineer.
- Standardized a set of common classmethods that are available to all plugins - `handle_settings` and `get_logger`.
- Updated bundled less.js to version 1.3.1.
- Lots of bug fixes.

version 0.3.2 - August 18, 2012

- Fixes a bug in the Markdown filter (used in *Template Pages*) that caused incorrect Markdown processing if there is leading white space in the Markdown content.
- Add table styles to included themes.

version 0.3.1 - August 5, 2012

- Fixes a rather nasty bug that would cause a fatal exception if there were non-ASCII characters in a post using the *Teaser Content* (post breaks) support that was added in version 0.3.0.
- Minor style fixes to Dark Rainbow theme.

version 0.3.0 - July 22, 2012

Important: The *theme plugin model* has changed with version 0.3.0. Installable themes will need to be changed to be compatible with the new model.

- A new *plugin model* provides a more flexible way to integrate with Engineer.
- Posts can now have *custom metadata*.
- New *Teaser Content* (post breaks) support.
- A sitemap is now generated automatically.
- A custom RSS feed url can be specified using the *FEED_URL* setting.
- Both *Dark Rainbow* and *Ole B* now include next/previous post links.

- Site-relative URLs for posts are now included in the post metadata during post normalization. This is useful in some cases where you need to know the URL of a post (for example, to link to it in another post) but are offline or otherwise unable to get the URL. If you put a manual URL in the post metadata, it will be overwritten - it's not used to actually allocate a URL for the post.
- Post metadata now accepts either `via-link` or `via_link`. Normalized metadata will now use `via-link` instead of `via_link` since the former feels more natural in YAML.
- The build process will now output a warning if there are pending posts in the site and *`PUBLISH_PENDING`* is `False`. This should help remind users that don't run a build automatically that they will need to run another build at a later date/time if they want the pending post to actually become visible.
- Bundled libraries updated:
  - LESS: version 1.3.0
  - jQuery: version 1.7.1
  - modernizr: version 2.5.3
- Themes can now indicate whether they use the bundled Tweet library by setting the *use_tweet* property.
- Fixed bug preventing some *Template Fragments* from being included properly in some themes.
- The included *Development server* no longer restricts requests to those coming from the same machine.
- Various build performance enhancements.
- Several fixes to bundled theme styles, including better mobile styles in Dark Rainbow.

version 0.2.4 - May 27, 2012

- A new theme, *Ole B*, has been added. This theme is based on Ole Begemann's oleb.net design and was created with his permission.
- During rendering, a new variable called `all_posts` is passed. It is a *`PostCollection`* containing all the posts on the site and can be used to display links to related posts, similarly tagged posts, etc.
- Themes can now be wrapped in a Python package, installed, and register themselves as a *theme plugin*.
- Bug fixes related to sites hosted at non-root paths.

version 0.2.3 - May 6, 2012

- External themes are now supported. You can place your custom theme either inside a `themes` directory in your site's root directory or in any directory you'd like using the *`THEME_DIRS`* setting.
- Themes can now specify *settings defaults* in their manifest.
- *Zipped themes* are now supported.
- Multiple *`verbosity levels`* are supported by the command line script now.
- *engineer serve* now supports a *`--port`* option.
- Build logs are now always written to a `build.log` file in the `logs` directory.
- CSS/JS compression process is now more efficient.
- Miscellaneous logging and cache fixes.

version 0.2.2 - April 30, 2012

- Updated sample site to disable `PREPROCESS_LESS` by default. This way the site will still build even if you don't have lessc installed or aren't on Windows.

version 0.2.1 - April 28, 2012

- Fixed corrupted LESS files that made it into v0.2.0.
- Fixed bug that prevented attribution text and links from showing up in Dark Rainbow theme.

version 0.2.0 - April 22, 2012

- Better post timezone handling.
- Various fixes to Dark Rainbow theme.
- Various fixes to the post cache mechanisms.
- Preprocessing support for LESS.
- Minification support for JS and CSS static files.
- New commands - 'clean' and 'init'.
- Major documentation improvements. (In other words, there is now documentation.)

version 0.1.0 - March 13, 2012

- Initial release.

CHAPTER 3

Developer Documentation

The Build Pipeline

Engineer goes through a number of steps when building a site. Understanding this order and the steps that are taken might be helpful when building your site. At the very least, it might be interesting.

Basic Flow

Engineer operates almost entirely on something called the 'output cache.' This is a location in the *CACHE_DIR*. Engineer essentially 'stages' all content there during processing. Once that's all done, then it synchronizes the actual *OUTPUT_DIR* with the cache.

With that in mind, these are the basic steps that Engineer goes through. You might also find it interesting to look at the code for the *engineer.commands.bundled.BuildCommand()* class - it is the primary entry point for the build process.

1. **Copy base Engineer static content**

   First, any static content from the core Engineer libraries is copied to the output cache. This includes things like the built-in jQuery, Modernizr, and Foundation libraries.

2. **Copy theme static content**

   Any static content needed by the theme is copied to the output cache.

3. **Generate template pages**

   Template Pages are generated and copied to the output cache.

4. **Load posts**

   Posts are loaded from the post cache and *POST_DIR* and stored in memory in a *PostCollection*.

5. **Generate posts**

   Based on settings like *PUBLISH_DRAFTS* and *PUBLISH_PENDING*, the list of posts is trimmed to those that should be output, then they are rendered and placed in the output cache.

6. **Generate rollup pages**

   The front page and other 'rollup' pages are generated and placed in the output cache.

7. **Generate archive page**

   The archive page is generated and placed in the output cache.

8. **Generate tag pages**

   Tag pages are generated and placed in the output cache.

9. **Generate feed and sitemap**

   The RSS feed and sitemap are generated and placed in the output cache.

10. **Copy raw content to output cache**

    Any site-specific *Raw Content* is copied to the output cache.

11. **Compress/minify CSS/JS**

    If the *COMPRESSOR_ENABLED* setting is on, then any CSS or JS files used in the site will be compressed.

    See also:

    CSS/JS Compression

12. **Remove source LESS files from output cache**

    If the *PREPROCESS_LESS* setting is on, then the source LESS files will be removed from the output cache. This is safe to do since the LESS preprocessing happens during rendering, so at this point the corresponding CSS file has already been generated. The LESS source file is no longer needed.

13. **Synchronize output directory with output cache**

    Finally, the contents of the *OUTPUT_DIR* is synchronized with the output cache. This approach ensures that the actual site output is disturbed as little as possible. All the major copying/generating/rendering has already happened separately on the output cache, so the actual site output directory has only changes/additions/deletions propagated to it.

Raw Content

Raw content is simply content that you want to include in your site as-is with no processing. Static content such as JavaScript and CSS could also be considered raw content but those sorts of files don't need to live at a specific place in your site's URL. The raw content feature is specifically for content where placement matters, such as for robots.txt files and favicons. These files must be in the root of your site, so just treating them like regular static content won't work.

Raw content should be placed in your site's *CONTENT_DIR*, and since it's the last thing copied in the build pipeline, it will overwrite any content that was generated by the other phases of the build pipeline. Keep this in mind.

The structure of *CONTENT_DIR* should match your site's. In other words, if you want something to wind up in the root of your site, you would put it at the root of your *CONTENT_DIR*. Similarly, if you want something to wind up in `/foo/` in your site, you'd put it in a `foo` folder inside your *CONTENT_DIR*.

CSS/JS Compression

TODO

LESS Preprocessing

TODO

Creating Your Own Themes

Theme Package Structure

Themes are essentially a folder with a manifest and a collection of templates and supporting static files (images, CSS, Javascript, etc.). Custom themes should be put in a `themes` folder within the site's root. You can put themes elsewhere by specifying the *THEME_DIRS* setting.

A sample theme folder might look like this:

```
.
- theme_id
|   - static
|   |   - scripts
|   |   |   - script.js
|   |   |   - ...
|   |   - stylesheets
|   |   |   - theme.less
|   |   |   - reset.css
|   |   |   - ...
|   |   - templates
|   |   |   - theme
|   |   |   |   - layouts
|   |   |   |   |   - ...
|   |   |   |   - _footer.html
|   |   |   |   - base.html
|   |   |   |   - post_list.html
|   |   |   |   - ...
|   - bundles.yaml
|   - metadata.yaml
```

Theme Manifest

Each theme must contain a file called `metadata.yaml` that contains metadata about the theme. The theme manifest is a simple text file in YAML format. The Dark Rainbow theme manifest looks like this, for example:

```
name: 'Dark Rainbow'
id: 'dark_rainbow'
description: 'A dark theme with just a hint of color.'
author: 'Tyler Butler <tyler@tylerbutler.com>'
website: 'http://tylerbutler.com'
license: 'Creative Commons BY-SA 3.0'
use_precompiled_styles: no

template_dirs:
  - '../_shared/templates/'

copy_content:
  - ['../_shared/images/rss/37.png', 'images/rss.png']
```

```
settings:
  typekit_id: ~
  comments: ~
  comments_account: ~
  simple_search: yes
  bigfoot:
    enabled: yes
  sharing:
    enabled: no
    facebook:
      enabled: no
      app_id: ~
    twitter:
      enabled: no
      username: ~
```

Theme Manifest Parameters

name The verbose human-readable name of the theme.

id The ID of the theme. This must match the folder name of the theme and should not contain spaces. This is used internally by Engineer to identify the theme.

self_contained ***(optional)*** Indicates whether the theme is self-contained or not. Defaults to `True` if not specified.

> **Note:** This parameter is not currently used and may be deprecated in the future.

description ***(optional)*** A more verbose description of the theme.

author ***(optional)*** The name and/or email address of the theme's author.

website ***(optional)*** The website where the theme or information about it can be found.

license ***(optional)*** The license under which the theme is made available.

use_precompiled_styles ***(optional)*** Indicates whether to use precompiled stylesheets. Defaults to `True`.

See also:

Static Content

New in version 0.5.0.

use_foundation ***(optional)*** Indicates whether the theme makes use of the Foundation CSS library included in Engineer. Defaults to `False`.

Deprecated since version 0.6.0: This setting is obsolete and ignored. Themes should now use

use_jquery ***(optional)*** Indicates whether the theme makes use of the jQuery library included in Engineer. Defaults to `False`.

use_lesscss ***(optional)*** Indicates whether the theme makes use of the LESS CSS library included in Engineer. Defaults to `False`.

use_modernizr ***(optional)*** Indicates whether the theme makes use of the Modernizr library included in Engineer. Defaults to `True`.

Changed in version 0.5.0: This setting now defaults to `True` instead of `False`.

use_normalize_css ***(optional)*** Indicates whether the theme makes use of the normalize.css file included in Engineer. Defaults to `True`.

New in version 0.5.0.

use_tweet ***(optional)*** Indicates whether the theme makes use of the Tweet library included in Engineer. Defaults to `False`.

Deprecated since version 0.5.0: This setting is obsolete and ignored. The Tweet library has been removed from Engineer. See the *Release Notes* for more information.

settings ***(optional)*** A dictionary of all the themes-specific settings that users of your theme can provide via *THEME_SETTINGS* and their default values. If your theme supports custom settings, you **must** specify defaults. Due to the way Engineer loads your theme settings and a user's site settings, your settings may not be created at all unless you specify them here.

New in version 0.2.3.

See also:

Referring to Custom Theme Settings in Templates

template_dirs ***(optional)*** A list of paths, each relative to the path to the theme manifest file itself, that should be included when searching for theme templates. These paths are in addition to the `templates` folder within the theme's folder itself, and will be searched in the order specified *after* the theme's `templates` folder.

Like *copy_content*, this parameter is useful if you are creating multiple themes that share common templates. You can specify the paths to the common templates and they will be available during the build process.

```
template_dirs:
  - '../_shared/templates/'
```

New in version 0.4.0.

copy_content ***(optional)*** A list of paths to files or directories that should be copied to the theme's output location during a build. This is useful if you are creating multiple themes that all share some common static content (JavaScript files, images, etc.). By specifying this parameter, content will be copied to a central location for you during the build process so you can include it in your theme templates, LESS files, etc.

Tip: Since Engineer uses webassets to manage static content in version 0.6.0+, the `copy_content` setting should be used for content other than CSS and JavaScript files, such as images or web fonts. Style and script files should be managed using *webassets bundles*.

This parameter should be a 'list of lists.' Each entry in the list is a list itself containing two items. The first item is the path to the file or folder that should be copied. *This path should be relative to the location of the theme manifest.*

The second parameter should be the target location for the file or folder. *The target path should be relative to the static/theme folder in the output folder.*

For example, consider the following `copy_content` parameter in a theme manifest:

```
copy_content:
  - ['../Font-Awesome-More/font', 'font']
  - ['../bootswatch/img/', 'img']
  - ['../bootstrap/js/', 'js']
```

In this example, the `../Font-Awesome-More/font` (a path relative to the location of the theme manifest file itself) will be copied to `static/theme/font`.

New in version 0.4.0.

Static Content

Starting with Engineer version 0.6.0, theme static content is managed using webassets. While themes can link directly to static content in their templates, using webassets is preferred. Webassets handles combining and minifying static content automatically.

New in version 0.6.0.

Bundles

Webassets uses 'bundles' to combine and minify static files. Bundles are essentially a collection of static files, as well as a set of filters that define what happens to the files (compiled to CSS/JS, minified, etc.), and an output file location.

Engineer *includes some pre-defined bundles*, which you can use in addition to defining your own. Bundles can include other bundles, so it's easy to include the pre-defined bundles into your own if you wish. This also makes it easy to share common static content across multiple themes.

In order to define your own bundles for your theme, create a YAML file called `bundles.yaml` alongside your *Theme Manifest*. This file should contain all the bundles used in your theme. Note that while webassets itself supports defining bundles directly in Python code, Engineer currently only uses YAML input for custom theme bundles.

The bundle format itself is straightforward. As an example, this is the `bundles.yaml` file for the *Dark Rainbow* theme:

```
dark_rainbow_css:
  filters: less, cssmin
  contents:
    - foundation6_css
    - './static/stylesheets/dark_rainbow.less'
  output: dark_rainbow.%(version)s.css

dark_rainbow_css_bigfoot:
  filters: less, cssmin
  contents:
    - bigfoot_css
    - './static/stylesheets/bigfoot.less'
    - dark_rainbow_css
  output: dark_rainbow_bigfoot.%(version)s.css
```

Note that all the paths to files, both input and output, should be relative to the `bundles.yaml` file itself. Make sure that your output file name includes the version placeholder (`%(version)s`) for cache-busting purposes. See URL Expiry (cache busting) for more details.

Also, keep in mind that while all of the filters supported by webassets are available for you to use in your bundles, many of them require external dependencies that are not included in Engineer by default. If you don't wish to require additional dependencies beyond what is included in Engineer, you should use only the `cssmin`, `jsmin`, and `less` filters. You can read more about all the webassets filters in the webassets documentation.

Included Bundles

Engineer includes several CSS/JavaScript libraries that you can use in your themes. These libraries are exposed as bundles. Simply add the appropriate bundle's name to the `contents` of your own bundle.

```
jquery:
  contents:
    - './jquery-1.11.0.min.js'
  output: 'jquery.%(version)s.js'

less:
  contents:
    - './less-2.5.3.min.js'
  output: 'less.%(version)s.js'

modernizr:
  contents:
    - './modernizr-2.7.1.min.js'
  output: 'modernizr.%(version)s.js'

foundation2_css:
  contents:
    - './foundation/stylesheets/grid.css'
    - './foundation/stylesheets/mobile.css'
  output: 'foundation.%(version)s.css'
  filters: cssmin

foundation2_css_ie:
  contents:
    - './foundation/stylesheets/ie.css'
  output: 'foundation_ie.%(version)s.css'
  filters: cssmin

foundation2_js:
  contents:
    - './foundation/javascripts/foundation.js'
  output: 'foundation.%(version)s.js'
  filters: jsmin

foundation6_css:
  contents:
    - './foundation6/css/foundation.css'
  output: 'foundation6.%(version)s.css'
  filters: cssmin

foundation6_js:
  contents:
    - './foundation6/js/foundation.min.js'
  output: 'foundation6.%(version)s.js'

normalize:
  contents:
    - './normalize/normalize.css'
  output: 'normalize.%(version)s.css'
  filters: cssmin

bigfoot_css:
  contents:
    - './bigfoot/dist/bigfoot-default.css'
  output: 'bigfoot.%(version)s.css'
  filters: cssmin

bigfoot_js:
```

```
contents:
  - './bigfoot/dist/bigfoot.js'
output: 'bigfoot.%(version)s.js'
filters: jsmin
```

Stylesheets

In addition to CSS, Engineer can automatically compile LESS stylesheets during a site build, so you are free to use LESS rather than CSS for your styles. When linking to your LESS stylesheet in your templates, you should use the `render_less_link` *macro*. This will ensure that the stylesheet is compiled as part of the site build process if needed.

Starting with Engineer 0.5.0, themes can include a 'precompiled' version of the LESS stylesheets they need. This is useful since in most cases users of your theme will not be making modifications to your LESS files. Thus, referencing a pre-built version of the stylesheet makes for faster builds.

In order to include a precompiled version of your stylesheets, simply add it alongside your regular stylesheet and append `_precompiled` to the name. For example, if your stylesheet is called `dark_rainbow.less`, then your precompiled version should be called `dark_rainbow_precompiled.css`. As long as you are referencing your stylesheet from your templates using the `render_less_link` *macro*, the precompiled version will automatically be picked up during a site build. No other changes are needed.

Required Templates

The following templates must be present in a theme's `templates/theme` folder:

- _single_post.html
- base.html
- post_archives.html
- post_detail.html
- post_list.html
- template_page_base.html

You can of course include additional templates or template fragments, for use either internally in your theme, or that users of your theme can take advantage of to further customize their site.

You should also ensure that your theme templates load the *Built-in Template Fragments* that Engineer users will expect.

Images

The built-in `img` function will output content using a template fragment at `templates/theme/_img.html`. Individual themes can override this output by providing their own custom template.

The template is always passed the following keyword variables:

- source
- classes
- width
- height
- title

- alt
- link

All except the `source` parameter are optional so the template should handle these cases appropriately. The default template looks like this:

```
{% if title %}
    <div class="image caption{% if classes %} {{ classes|join(' ') }}{% endif %}">
{% else %}
    <div class="image{% if classes %} {{ classes|join(' ') }}{% endif %}">
{% endif %}

{% if link %}
    <a href="{{ link }}"><img src="{{ source|trim }}"
                              {%- if width %} width="{{ width|trim }}"{%- endif %}
                              {%- if height %} height="{{ height|trim }}"{%- endif %}
                              {%- if alt %} alt="{{ alt|trim }}"{%- endif %}
                              {%- if title %} title="{{ title|trim }}"{%- endif %}/>
    </a>
{% else %}
    <img src="{{ source|trim }}"
         {%- if width %} width="{{ width|trim }}"{%- endif %}
         {%- if height %} height="{{ height|trim }}"{%- endif %}
         {%- if alt %} alt="{{ alt|trim }}"{%- endif %}
         {%- if title %} title="{{ title|trim }}"{%- endif %}/>
{% endif %}

{% if title %}
    <p>{{ title }}</p>
{% endif %}

</div>
```

See also:

Including Images

New in version 0.5.0.

Sitemap Templates

Themes can provide custom templates for sitemap, just as *individual sites can*. These templates should be in the theme's `templates/theme` folder.

New in version 0.3.0.

Referring to Custom Theme Settings in Templates

Custom theme settings are available in all Engineer templates. Every template is passed a context variable called `theme` that represents the current theme. Any custom settings specified are available as attributes on that object. For example, if your theme defines a custom setting called `typekit_id`, then you can refer to that setting in any Engineer template like so:

```
{# TYPEKIT #}
<script type="text/javascript"
        src="http://use.typekit.com/{{ theme.typekit_id }}.js"></script>
<script type="text/javascript">
```

```
    try {
        Typekit.load();
    } catch (e) {}
</script>
```

Useful Macros

TODO

Zipping Themes

You can optionally put your theme directory in a zip file. The file should have a `.zip` file extension. Engineer will unzip the folder to a temporary location during a build and load the theme from that temporary location.

New in version 0.2.3.

Sharing Your Theme

The simplest way to share your theme is to *zip it up* and make it available to download. Users can then download it and use it by placing it in their site's `themes` directory or in another one of their *THEME_DIRS*.

You may wish instead to deliver your theme as an installable Python package. This allows users to download and install your theme via `pip` or any other tool. See *Theme Plugins* for more details.

Add Your Theme to Engineer

If you'd like to make your theme available with the main Engineer application, send a pull request on github.

Plugins

Engineer provides a plugin model that allows further customization of Engineer behavior. While Engineer contained a rudimentary plugin system for themes in version 0.2.3, version 0.3.0 introduced a much richer system and exposed ways to modify the post rendering pipeline somewhat in addition to Theme plugins.

Engineer's plugin model is based on Marty Alchin's simple plugin framework. As such, creating your own plugin is relatively straightforward:

1. Subclass one of the available plugin base classes (e.g. *PostProcessor*)
2. Load the module containing your plugin during an Engineer operation

Step 1 is quite simple. Step 2 is slightly more involved, but you have a couple of options for your plugin.

Loading Plugins

In order for plugins to be found, the module containing them must be imported by Engineer. Engineer provides two ways to achieve this. First, you can use the *PLUGINS* setting. Each module passed in via that setting will be imported, and the plugins they contain will be available to Engineer.

Tip: Command plugins are the exception; they cannot be loaded using the *PLUGINS* setting. They must be installed as a Python package using the method below.

Alternatively, you can deliver your plugin as an installable Python package. This allows users to download and install your theme via `pip` or any other tool. You can do this by adding an `engineer.plugins` setuptools entry point to your `setup.py` file.

In particular, within the *setup* function call in your `setup.py` file, add something like the following:

```
entry_points={
    'engineer.plugins': ['post_processors=dotted.path.to.module',
                         'themes=another.module.path'],
    }
```

The above code registers two plugin modules with Engineer. Engineer will import these modules, and any subclasses of the plugin base classes will be automatically discovered and run with Engineer.

The identifiers to the left of the equals sign (i.e. `post_processors` and `themes`) can be anything at all. Engineer doesn't look at them or use them. For clarity, in the above example the plugins have been broken into different modules by type, and each module has an identifier based on the type of plugin that module contains. But again, this is not required. It could just as easily read:

```
['foo=dotted.path.to.module',
 'bar=another.module.path']
```

The type of the plugin is determined by its parent class, not by its module or a specific identifier in the setup function.

Tip: The *only* requirement to get your plugin loaded is for the module containing it to be imported. Thus, if you have a number of plugins in different modules, you could create a wrapper module that simply imported the others, then sent your entry point to point to the wrapper module. When the wrapper module is imported, the other modules will also be imported, and then your plugins will be magically loaded.

Plugin Permissions

Some plugin capabilities are restricted and require explicit permission from the Engineer user via the *PLUGIN_PERMISSIONS* setting. As of Engineer 0.5.0 there is only one permission available, `MODIFY_RAW_POST`.

Prior to Engineer 0.5.0, it was not possible for plugins to modify actual post content. The *Metadata Finalization* plugin modified post metadata, but post content itself was never changed. This was a deliberate design decision to try and prevent data loss from runaway plugins. It was especially helpful during plugin testing when bugs weren't yet found and fixed.

In Engineer 0.5.0, plugins can now modify post content using the *set_finalized_content()* method on the Post class. However, this is protected by the `MODIFY_RAW_POST` permission. If the plugin is not explicitly listed as having that permission in the user's config, then calls to `set_finalized_content` will do nothing.

Note: The plugin permissions system is a little clunky and overly protective. The intent of the system is to help prevent plugins from doing potentially damaging things (like editing post source content) without explicit permission from the user. However, it's possible that I'm being paranoid and that this is overkill. Thus, consider this 'experimental' in Engineer 0.5.0. It may go away in the future; I welcome feedback on this.

New in version 0.5.0.

Common Plugin Methods

All plugins inherit the some methods from *PluginMixin*. Note that you should not subclass the mixin yourself; rather, you should subclass one of the relevant plugin base classes below. The *PluginMixin* class is documented only for completeness.

class engineer.plugins.core.**PluginMixin**

classmethod get_logger(*custom_name=None*)
Returns a logger for the plugin.

classmethod handle_settings(*config_dict*, *settings*)
If a plugin defines its own settings, it may also need to handle those settings in some unique way when the Engineer configuration files are being read. By overriding this method, plugins can ensure such unique handling of their settings is done.

Note that a plugin does not have to handle its own settings unless there is unique processing that must be done. Any settings that are unknown to Engineer will automatically be added as attributes on the *EngineerConfiguration* object. This method should only be implemented if the settings must be processed in some more complicated way prior to being added to the global configuration object.

Implementations of this method should check for the plugin-specific settings in `config_dict` and set appropriate attributes/properties on the `settings` object. In addition, settings that have been handled should be removed from `config_dict`. This ensures they are not handled by other plugins or the default Engineer code.

Parameters

- **config_dict** – The dict of as-yet unhandled settings in the current settings file.
- **settings** – The global *EngineerConfiguration* object that contains all the
- **settings** – The global *EngineerConfiguration* object that contains all the settings for the current Engineer process. Any custom settings should be added to this object.

Returns The modified `config_dict` object.

Jinja Environment Plugins

class engineer.plugins.**JinjaEnvironmentPlugin**
Bases: *engineer.plugins.core.PluginMixin*

Base class for JinjaEnvironment *Plugins*.

JinjaEnvironment plugins can supplement the Jinja 2 environment with things like filters and global functions. These additions can then be used in your Jinja templates.

New in version 0.5.0.

classmethod get_filters()
If required, subclasses can override this method to return a dict of filters to add to the Jinja environment. The default implementation simply returns *filters*.

classmethod get_globals()
If required, subclasses can override this method to return a dict of functions to add to the Jinja environment globally. The default implementation simply returns *globals*.

classmethod **update_environment** (*jinja_env*)
For complete customization of the Jinja environment, subclasses can override this method.

Subclasses should ensure that the base implementation is called first in their overridden implementation. For example:

```
@classmethod
def update_environment(cls, jinja_env):
    super(BundledFilters, cls).update_environment(jinja_env)
    # some other code here...
```

Parameters **jinja_env** – The Jinja environment.

filters = {}
A dict of filters to add to the Jinja environment. The key of each entry should be the name of the filter (as it will be used inside templates), while the value should be the filter function. If you require more custom logic to build the dict of filters, override the *get_filters()* method.

globals = {}
A dict of functions to add to the Jinja environment globally. The key of each entry should be the name of the function (as it will be used inside templates), while the value should be the function itself. If you require more custom logic to build this dict, override the *get_globals()* method.

Post Processor Plugins

class engineer.plugins.**PostProcessor**
Bases: *engineer.plugins.core.PluginMixin*

Base class for Post Processor *Plugins*.

PostProcessor subclasses should provide implementations for *preprocess()* or *postprocess()* (or both) as appropriate.

classmethod **postprocess** (*post*)
The postprocess method is called after the post has been imported and processed as well as converted to HTML and output.

Parameters **post** – The post being currently processed by Engineer.

Returns The *post* parameter should be returned.

classmethod **preprocess** (*post*, *metadata*)
The preprocess method is called during the Post import process, before any post metadata defaults have been set.

The preprocess method should use the content_preprocessed attribute to get/modify the content of *post*. This ensures that preprocessors from other plugins can be chained together.

By default, the content_preprocessed value is used only for generating post HTML. It is not written back to the source post file. However, sometimes you may want to make a permanent change to the post content that is written out. In this case, you should call the *set_finalized_content()* method, passing it the modified content. This method will ensure the data is written back to the source file by the *Metadata Finalization* plugin. This means that in order for a plugin to write preprocessed data back to the post file, the *FINALIZE_METADATA* setting must be enabled.

Your plugin will also need to be explicitly granted the MODIFY_RAW_POST permission. See more detail in *Plugin Permissions*.

In addition, the preprocess method can add/remove/update properties on the *post* object itself as needed.

Tip: Since the *FINALIZE_METADATA* setting must be enabled for plugins to write back to source post files, you should check this setting in addition to any other settings you may be using.

Parameters

- **post** – The post being currently processed by Engineer.
- **metadata** – A dict of the post metadata contained in the post source file. It contains no default values - only the values contained within the post source file itself. The preprocess method can add, update, or otherwise manipulate metadata prior to it being processed by Engineer manipulating this parameter.

Returns The *post* and *metadata* values should be returned (as a 2-tuple) by the method.

Theme Plugins

class engineer.plugins.**ThemeProvider**

Bases: *engineer.plugins.core.PluginMixin*

Base class for Theme *Plugins*.

ThemeProvider subclasses must provide a value for *paths*.

Changed in version 0.3.0.

paths = ()

An iterable of absolute paths containing one or more *theme manifests*.

The actual Python code needed to register your theme as a plugin is very minimal, but it is overhead compared to simply downloading a theme directly. The benefit, of course, is that users can manage the installation of the theme alongside Engineer itself, and since the theme is globally available, users don't need to copy the theme to each site they want to use it in.

Command Plugins

The Engineer command line can be customized to include your own commands. See *Command Plugins* for more information.

Changed in version 0.6.0: Command plugins changed dramatically in version 0.6.0 and are now *documented separately*.

Command Plugins

There are many cases where it may be useful to enhance Engineer by adding new commands to its *command line interface*. Engineer provides a way for you to do this fairly easily using the same model as *other plugins*. In fact, the core Engineer commands are implemented as plugins that come bundled with Engineer.

Tip: Unlike other plugin types, Command plugins cannot be loaded using the *PLUGINS* setting. They must be installed as a Python package and are therefore available to all sites using a given Engineer installation.

Command Plugin Types

Engineer provides two forms of command extensibility: argparse and argh. The argparse style will be familiar to anyone who has used argparse, the command-line processing module in the Python standard library.

The alternative is to use argh, which is a wrapper around argparse that provides a great deal of simplifying functionality while retaining the power inherent in argparse.

So, what style should you use? In general, I recommend using argh if you're new to Python or command-line handling in general. I think it's much simpler to use, and it requires a lot less boilerplate code. While none of the core Engineer commands use argh, there are examples below that will guide you through the process.

One very minor drawback to argh is that as of Engineer version 0.6.0, it is not included as a dependency and thus is not installed by default. This means that you'll need to include it as a dependency in your own plugin's package. An upcoming version of Engineer may include it so this would no longer be necessary.

The core Engineer commands were originally written before the command plugin model existed, and were written to use argparse directly. Depending on what you're doing you might find it more powerful. Certainly if you already have experience with argparse, there's no reason to go out and learn how to use argh unless you want to.

Basic Plugin Model

As with *other plugin types*, command plugins are implemented by subclassing a plugin base class. Unlike other plugins, however, there are multiple base classes to use. In addition, there is a more complex class hierarchy including some private mixin classes that are documented here for completeness, but that you shouldn't need to subclass directly in your plugins.

The mixins and base classes abstract away most of the complexity of dealing with the guts of the parsers, and provide simple ways to plug in your own functions. In addition, you can also add the `verbose` and `settings` options that are available in most Engineer commands easily without implementing them yourself.

Engineer's command processing is built using argparse. One of argparse's 'quirks,' or design decisions/constraints, is that it is not easy to access subparsers arbitrarily. Essentially, you can only parsers very early on in the process of building the argparse objects. Thus, all command plugins are passed a `main_parser`, which is the main ArgumentParser object, when they are instantiated. In addition, they are passed the top-most `subparser` object, created by initially calling `add_subparsers` on the main ArgumentParser object. With these two components, it is possible to manipulate commands in diverse ways.

Fortunately, for the most part, plugin implementers needn't be concerned with this detail, since it is abstracted away by various subclasses.

Tip: It is easiest to model each of your commands as a single independent class wherever possible. In many cases, this will be straightforward. If, however, you want to add a more complicated command, or a command with subcommands and argh, you can do this. See the examples below.

Argparse-based Plugins

In order to implement an argparse-based plugin, you should subclass `ArgparseCommand`.

class `engineer.commands.core.`**`ArgparseCommand`**(*main_parser*, *top_level_parser=None*)

Bases: `engineer.commands.core._ArgparseMixin`

Serves as a base class for simple argparse-based commands. All built-in Engineer commands, such as *engineer clean*, are examples of this type of command. See the source for the classes in the `engineer.commands.bundled` module for a specific example.

get_logger(*custom_name=None*)
Returns a logger for the plugin.

handle_settings(*config_dict*, *settings*)
If a plugin defines its own settings, it may also need to handle those settings in some unique way when the Engineer configuration files are being read. By overriding this method, plugins can ensure such unique handling of their settings is done.

Note that a plugin does not have to handle its own settings unless there is unique processing that must be done. Any settings that are unknown to Engineer will automatically be added as attributes on the *EngineerConfiguration* object. This method should only be implemented if the settings must be processed in some more complicated way prior to being added to the global configuration object.

Implementations of this method should check for the plugin-specific settings in `config_dict` and set appropriate attributes/properties on the `settings` object. In addition, settings that have been handled should be removed from `config_dict`. This ensures they are not handled by other plugins or the default Engineer code.

Parameters

- **config_dict** – The dict of as-yet unhandled settings in the current settings file.
- **settings** – The global *EngineerConfiguration* object that contains all the
- **settings** – The global *EngineerConfiguration* object that contains all the settings for the current Engineer process. Any custom settings should be added to this object.

Returns The modified `config_dict` object.

handler_function(*args=None*)
This function contains your actual command logic. Note that if you prefer, you can implement your command function with a different name and simply set `handler_function` to be the function you defined. In other words:

```
def my_function(*args, **kwargs):
    # my implementation
    pass

handler_function = my_function
```

The built-in Engineer commands all use this approach. You can see the source for those classes in the *engineer.commands.bundled* module.

help
The help string for the command.

name
The name of the command.

need_settings
Defaults to True. Set to False if the command does not require an Engineer config file.

need_verbose
Defaults to True. Set to False if the command does not support the standard Engineer *verbose* option.

parser
Returns the appropriate parser to use for adding arguments to your command.

Argh-based Plugins

More Advanced Plugin Styles

class engineer.commands.core.**Command**(*main_parser*, *top_level_parser=None*)

The most barebones command plugin base class. You should use *ArgparseCommand* or ArghCommand wherever possible.

Macros

TODO

```
{% macro theme_local(path) -%}

{%- endmacro %}

{%- macro _make_relative_path(path, prepend_static_url) -%}
    {%- if prepend_static_url -%}
        {{ STATIC_URL }}/{{ path }}
    {%- else -%}
        {{ path }}
    {%- endif -%}
{%- endmacro -%}

{%- macro render_less_link(path_input, prepend_static_url=True) -%}
    {%- if theme.use_precompiled_styles %}
        {% filter compress %}
            <link rel="stylesheet" href="{{ _make_relative_path(make_precompiled_
↪reference(path_input), prepend_static_url) }}"
                  type="text/css"/>
        {% endfilter %}
    {%- elif not settings.PREPROCESS_LESS -%}
        <link rel="stylesheet/less" href="{{ _make_relative_path(path_input, prepend_
↪static_url) }}" type="text/css"/>
    {%- else -%}
        {% if path_input.endswith('.less') %}
            {{ preprocess_less(path_input) }}
            {% set path_input=("%s.css" % path_input[:-5]) %}
        {% endif %}

        {% filter compress %}
            <link rel="stylesheet" href="{{ _make_relative_path(path_input, prepend_
↪static_url) }}" type="text/css"/>
        {% endfilter %}
    {%- endif -%}
{%- endmacro -%}

{% macro render_script_link(path_input, prepend_static_url=True) -%}
    {% filter compress %}
        <script src="{{ _make_relative_path(path_input, prepend_static_url) }}"
                type="text/javascript">
        </script>
    {% endfilter %}
{%- endmacro -%}

{% macro navigation_link(name, url, section, active_class=None) -%}
```

```
    {% if active_class is none %}
        {% set active_class=settings.ACTIVE_NAV_CLASS %}
    {% endif %}
    <li{% if nav_context in section %} class="{{ active_class }}"{% endif %}>
        <a href="{{ url }}">{{ name }}</a></li>
{%- endmacro %}
```

CHAPTER 4

API Documentation

engineer.commands.bundled

class engineer.commands.bundled.**BuildCommand**(*main_parser*, *top_level_parser=None*)
Builds an Engineer site.

See also:

engineer build

class engineer.commands.bundled.**CleanCommand**(*main_parser*, *top_level_parser=None*)
Cleans an Engineer site's output directory and clears all caches.

See also:

engineer clean

class engineer.commands.bundled.**InitCommand**(*main_parser*, *top_level_parser=None*)
Initializes a new engineer site in the current directory.

See also:

engineer init

class engineer.commands.bundled.**ServeCommand**(*main_parser*, *top_level_parser=None*)
Serves an Engineer site using a built-in development server.

See also:

engineer serve

engineer.conf

class engineer.conf.**EngineerConfiguration**(*settings_file=None*, *override=None*)
Stores all of the configuration settings for a given Engineer site.

This class uses the Borg design pattern and shares state among all instances of the class.

There seem to be a lot of differing opinions about whether this design pattern is A Good Idea (tm) or not. It definitely seems better than Singletons since it enforces *behavior*, not *structure*, but it's also possible there's a better way to do it in Python with judicious use of globals.

create_required_directories()
Creates any directories required for Engineer to function if they don't already exist.

engineer.engine

engineer.enums

class engineer.enums.**Status**
Enum representing the status of a *Post*.

draft = <EnumValue: Status.draft [value=0]>
Post is a draft.

published = <EnumValue: Status.published [value=1]>
Post is published.

review = <EnumValue: Status.review [value=2]>
Post is in review.

engineer.models

class engineer.models.**Post**(*source*)
Represents a post written in Markdown and stored in a file.

Parameters **source** – path to the source file for the post.

render_item(*all_posts*)
Renders the Post as HTML using the template specified in `html_template_path`.

Parameters **all_posts** – An optional *PostCollection* containing all of the posts in the site.

Returns The rendered HTML as a string.

set_finalized_content(*content*, *caller_class*)
Plugins can call this method to modify post content that is written back to source post files. This method can be called at any time by anyone, but it has no effect if the caller is not granted the `MODIFY_RAW_POST` permission in the Engineer configuration.

The *FINALIZE_METADATA* setting must also be enabled in order for calls to this method to have any effect.

Parameters

- **content** – The modified post content that should be written back to the post source file.
- **caller_class** – The class of the plugin that's calling this method.

Returns `True` if the content was successfully modified; otherwise `False`.

content
The post's content in HTML format.

content_template = None
The path to the template to use to transform the post *content* into HTML.

custom_properties = None
A dict of any custom metadata properties specified in the post.

is_draft
`True` if the post is a draft, `False` otherwise.

is_external_link
`True` if the post has an associated external link. `False` otherwise.

is_pending
`True` if the post is marked as published but has a timestamp set in the future.

is_published
`True` if the post is published, `False` otherwise.

link = None
The post's *external link*.

markdown_template_path = None
The path to the template to use to transform the post back into a *post source file*.

slug = None
The slug for the post.

source = None
The absolute path to the source file for the post.

status = None
The status of the post (published or draft).

template = None
The path to the template to use to transform the post into HTML.

timestamp = None
The date/time the post was published or written.

timestamp_local
The post's *timestamp* in 'local' time.

Local time is determined by the *POST_TIMEZONE* setting.

title = None
The title of the post.

updated = None
The date/time the post was updated.

via = None
The post's attribution name.

via_link = None
The post's attribution link.

class engineer.models.**PostCollection**(*seq=()*)
A collection of *Posts*.

tagged(*tag*)
Returns a new PostCollection containing the subset of posts that are tagged with *tag*.

all_tags
Returns a list of all the unique tags, as strings, that posts in the collection have.

drafts
Returns a new PostCollection containing the subset of posts that are drafts.

pending
Returns a new PostCollection containing the subset of posts that are pending.

published
Returns a new PostCollection containing the subset of posts that are published.

review
Returns a new PostCollection containing the subset of posts whose status is `review`.

engineer.themes

class `engineer.themes.`**`Theme`**(*theme_root_path*, ***kwargs*)
Creates a new theme object based on the contents of *theme_root_path*.

engineer.finders

class `engineer.themes.finders.`**`DefaultFinder`**
Locates and loads built-in Engineer themes.

class `engineer.themes.finders.`**`PluginFinder`**
Loads themes from any installed *Theme Plugins*.

New in version 0.2.4.

class `engineer.themes.finders.`**`SiteFinder`**
Loads themes from the `/themes` directory inside a site folder.

New in version 0.2.3.

class `engineer.themes.finders.`**`ThemeDirsFinder`**
Loads themes from the directories specified in *THEME_DIRS*.

New in version 0.2.3.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V