Flask-Blogging Documentation

Release 0.1.0

Gouthaman Balaraman

July 07, 2015

Contents

Flask-Blogging is a Flask extension for adding Markdown based blog support to your site. It provides a flexible mechanism to store the data in the database of your choice. It is meant to work with the authentication provided by packages such as Flask-Login or Flask-Security.

The philosophy behind this extension is to provide a lean app based on Markdown to provide blog support to your existing web application. This is contrary to some other packages such as Flask-Blog that are just blogs. If you already have a web app and you need to have a blog to communicate with your user or to promote your site through content based marketing, then Flask-Blogging would help you quickly get a blog up and running.

Out of the box, Flask-Blogging has support for the following:

- Bootstrap based site
- Markdown based blog editor
- Models to store blog
- Authentication of User's choice
- Sitemap, ATOM support
- Disqus support for comments
- Google analytics for usage tracking
- Well documented, tested, and extensible design

CHAPTER 1

Quick Start Example

```
from flask import Flask, render_template_string, redirect
from sqlalchemy import create_engine
from flask.ext.login import UserMixin, LoginManager, \
    login_user, logout_user
from flask.ext.blogging import SQLAStorage, BloggingEngine

app = Flask(__name__)
app.config["SECRET_KEY"] = "secret"  # for WTF-forms and login
app.config["BLOGGING_URL_PREFIX"] = "/blog"
app.config["BLOGGING_DISQUS_SITENAME"] = "test"
app.config["BLOGGING_SITEURL"] = "http://localhost:8000"

# extensions
engine = create_engine('sqlite:////tmp/blog.db')
sql_storage = SQLAStorage(engine)
blog_engine = BloggingEngine(app, sql_storage)
login_manager = LoginManager(app)

# user class for providing authentication
class User(UserMixin):
    def __init__(self, user_id):
        self.id = user_id

    def get_name(self):
        return "Paul Dirac"  # typically the user's name

@login_manager.user_loader
@blog_engine.user_loader
def load_user(user_id):
    return User(user_id)

index_template = """
<!DOCTYPE html>
<html>
    <head> </head>
    <body>
        {% if current_user.is_authenticated() %}
            <a href="/logout/">Logout</a>
        {% else %}
            <a href="/login/">Login</a>
        {% endif %}
          <a href="/blog/">Blog</a>
          <a href="/blog/sitemap.xml">Sitemap</a>
```

```
          <a href="/blog/feeds/all.atom.xml">ATOM</a>
    </body>
</html>
"""

@app.route("/")
def index():
    return render_template_string(index_template)

@app.route("/login/")
def login():
    user = User("testuser")
    login_user(user)
    return redirect("/blog")

@app.route("/logout/")
def logout():
    logout_user()
    return redirect("/")


if __name__ == "__main__":
    app.run(debug=True, port=8000, use_reloader=True)
```

The key components required to get the blog hooked is explained below.

CHAPTER 2

Configuring your Application

The *BloggingEngine* class is the gateway to configure blogging support to your web app. You should create the *BloggingEngine* instance like this:

```
blogging_engine = BloggingEngine()
```

You also need to pick the storage for blog. That can be done as:

```
from sqlalchemy import create_engine

engine = create_engine("sqlite:////tmp/sqlite.db")
storage = SQLAStorage(engine)
```

Once you have created the blogging engine and the storage, you can connect with your app using the *init_app* method as shown below:

```
blogging_engine.init_app(app, storage)
```

Flask-Blogging lets the developer pick the authentication that is suitable, and hence requires her to provide a way to load user information. You will need to provide a *BloggingEngine.user_loader* callback. This callback is used to load the user from the *user_id* that is stored for each blog post. Just as in Flask-Login, it should take the *unicode user_id* of a user, and return the corresponding user object. For example:

```
@blogging_engine.user_loader
def load_user(userid):
    return User.get(userid)
```

For the blog to have a readable display name, the `User` class must implement either the `get_name` method or the `__str__` method.

The `BloggingEngine` accepts an optional `extensions` argument. This is a list of `Markdown` extensions objects to be used during the markdown processing step.

The `BloggingEngine` also accepts `post_processor` argument, which can be used to provide a custom post processor object to handle the processing of Markdown text. An ideal way to do this would be to inherit the default `PostProcessor` object and override custom methods. There is a `custom_process` method that can be overridden to add extra functionality to the post processing step.

The `BloggingEngine` can be configured by setting the following app config variables. These arguments are passed to all the views. The keys that are currently supported include:

BLOGGING_SITENAME	The name of the blog to be used as the brand name. This is also used in the feed heading. (default "Flask-Blogging")
BLOGGING_SITEURL	The url of the site.
BLOGGING_RENDER_TEXT	Boolean value to specify if the raw text should be rendered or not. (default `True`)
BLOGGING_DISQUS_SITENAME	Disqus sitename for comments (default `None`)
BLOGGING_GOOGLE_ANALYTICS	Google analytics code for usage tracking (default `None`)
BLOGGING_URL_PREFIX	The prefix for the URL of blog posts (default `None`)
BLOGGING_FEED_LIMIT	The number of posts to limit to in the feed. If `None`, then all are shown, else will be limited to this number. (default `None`)

CHAPTER 3

Blog Views

There are various views that are exposed through Flask-Blogging. If the `url_prefix` argument in the BloggingEngine is `/blog`, then the URL for the various views are:

- `/blog/` (GET): The index blog posts with the first page of articles.
- `/blog/page/<post_id>/<optional slug>/` (GET): The blog post corresponding to the `post_id` is retrieved.
- `/blog/tag/<tag_name>/` (GET): The list of blog posts corresponding to `tag_name` is returned.
- `/blog/author/<user_id>/` (GET): The list of blog posts written by the author `user_id` is returned.
- `/blog/editor/` (GET, POST): The blog editor is shown. This view needs authentication.
- `/blog/delete/<post_id>/` (POST): The blog post given by `post_id` is deleted. This view needs authentication.
- `/blog/sitemap.xml` (GET): The sitemap with a link to all the posts is returned.

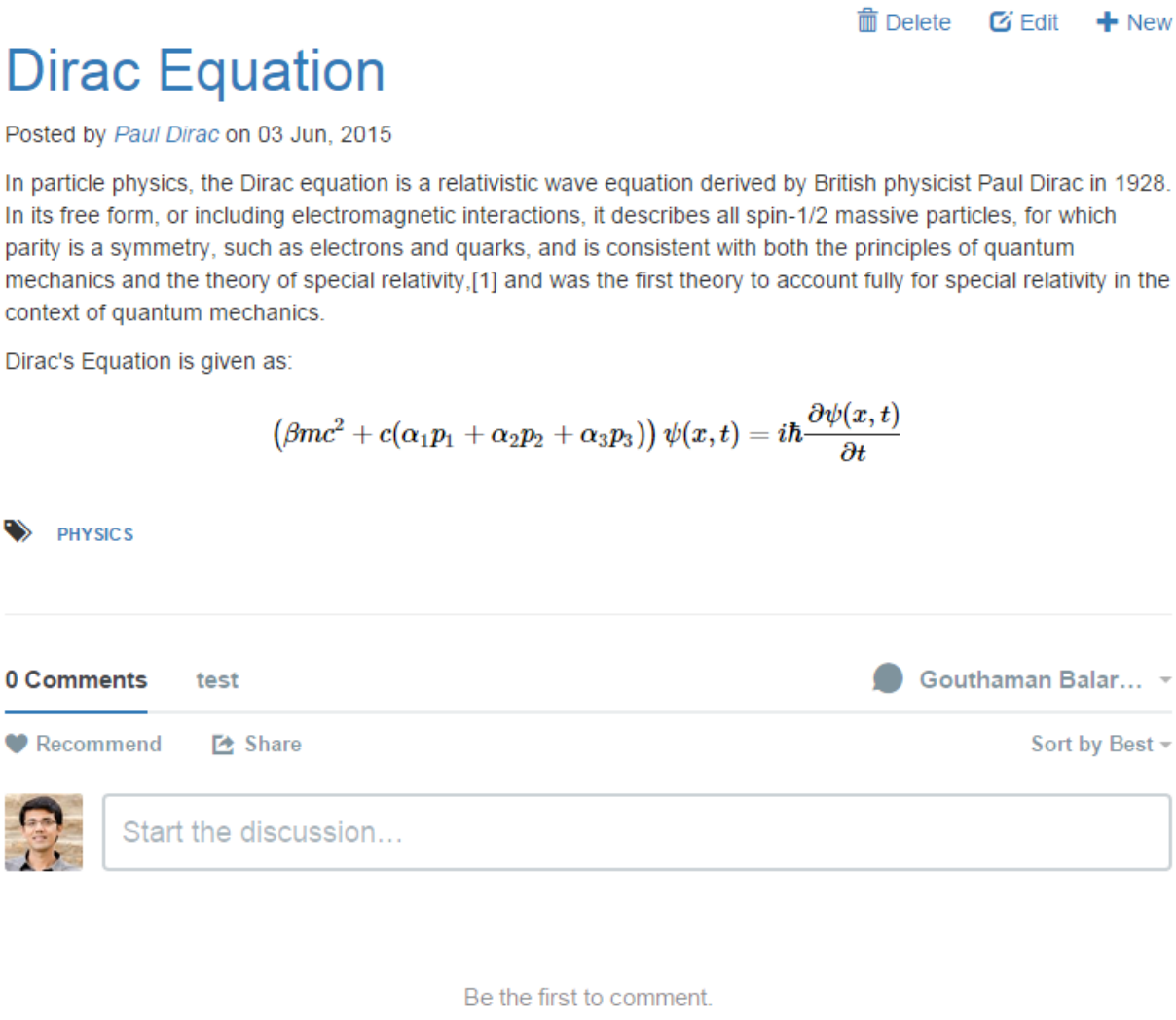
The view can be easily customised by the user by overriding with their own templates. The template pages that need to be customized are:

- `blog/index.html`: The blog index page used to serve index of posts, posts by tag, and posts by author
- `blog/editor.html`: The blog editor page.
- `blog/page.html`: The page that shows the given article.
- `blog/sitemap.xml`: The sitemap for the blog posts.

CHAPTER 4

Screenshots

4.1 Blog Page



4.2 Blog Editor

CHAPTER 5

Useful Tips

- **Postgres using psycopg2**: If you use `psycopg2` driver for Postgres while using the `SQLAStorage` you would need to have `autocommit` turned on while creating the engine:

```
create_engine("postgresql+psycopg2://postgres:@localhost/flask_blogging",
              isolation_level="AUTOCOMMIT")
```

- **Migrations with Alembic**: If you have migrations part of your project using Alembic, or extensions such as `Flask-Migrate` which uses Alembic, then you have to modify the `Alembic` configuration in order for it to ignore the `Flask-Blogging` related tables. If you don't set these modifications, then every time you run migrations, `Alembic` will not recognize the tables and mark them for deletion. And if you happen to `upgrade` by mistake then all your blog tables will be deleted. What we will do here is ask Alembic to `exclude` the tables used by `Flask-Blogging`. In your `alembic.ini` file, add a line:

```
[alembic:exclude]
tables = tag, post, tag_posts, user_posts
```

And in your `env.py`, we have to mark these tables as the ones to be ignored.

```
def exclude_tables_from_config(config_):
    tables_ = config_.get("tables", None)
    if tables_ is not None:
        tables = tables_.split(",")
    return tables

exclude_tables = exclude_tables_from_config(config.get_section('alembic:exclude'))

def include_object(object, name, type_, reflected, compare_to):
    if type_ == "table" and name in exclude_tables:
        return False
    else:
        return True

def run_migrations_online():
    """Run migrations in 'online' mode.

    In this scenario we need to create an Engine
    and associate a connection with the context.

    """
    engine = engine_from_config(
                config.get_section(config.config_ini_section),
                prefix='sqlalchemy.',
                poolclass=pool.NullPool)
```

```
        connection = engine.connect()
        context.configure(
                    connection=connection,
                    target_metadata=target_metadata,
                    include_object=include_object,
                    compare_type=True
                    )

        try:
            with context.begin_transaction():
                context.run_migrations()
        finally:
            connection.close()
```

In the above, we are using `include_object` in `context.configure(...)` to be specified based on the `include_object` function.

CHAPTER 6

Release Notes

- **Version 0.2.0**:

  Released July 6, 2015

  - `BloggingEngine` configuration moved to the `app` config setting. This breaks backward compatibility. See compatibility notes below.
  - Added ability to limit number of posts shown in the feed through `app` configuration setting.
  - The `setup.py` reads version from the module file. Improves version consistency.

- **Version 0.1.2**:

  Released July 4, 2015

  - Added Python 3.4 support

- **Version 0.1.1**:

  Released June 15, 2015

  - Fixed PEP8 errors
  - Expanded SQLAStorage to include Postgres and MySQL flavors
  - Added `post_date` and `last_modified_date` as arguments to the

  `Storage.save_post(...)` call for general compatibility

- **Version 0.1.0**:

  Released June 1, 2015

  - Initial Release
  - Adds detailed documentation
  - Supports Markdown based blog editor
  - Has 90% code coverage in unit tests

CHAPTER 7

Compatibility Notes

- **Version 0.2.0**:

  In this version, `BloggingEngine` will no longer take `config` argument. Instead, all configuration can be done through `app` config variables. Another `BloggingEngine` parameter, `url_prefix` is also available only through config variable.

CHAPTER 8

API Documentation

8.1 Module contents

8.2 Submodules

8.3 flask_blogging.engine module

The BloggingEngine module.

class flask_blogging.engine.**BloggingEngine**(*app=None*, *storage=None*, *post_processor=None*, *extensions=None*)

Bases: object

The BloggingEngine is the class for initializing the blog support for your web app. Here is an example usage:

```
from flask import Flask
from flask.ext.blogging import BloggingEngine, SQLAStorage
from sqlalchemy import create_engine

app = Flask(__name__)
db_engine = create_engine("sqlite:////tmp/sqlite.db")
storage = SQLAStorage(db_engine)
blog_engine = BloggingEngine(app, storage)
```

__init__(*app=None*, *storage=None*, *post_processor=None*, *extensions=None*)

Parameters

- **app** (*object*) – Optional app to use
- **storage** (*object*) – The blog storage instance that implements the Storage class interface.
- **post_processor** (*object*) – (optional) The post processor object. If none provided, the default post processor is used.
- **extensions** (*list*) – A list of markdown extensions to add to post processing step.

Returns

init_app(*app*, *storage*)

Initialize the engine.

Parameters

- **app** – The app to use
- **storage** – The blog storage instance that implements the `Storage` class interface.

user_loader(*callback*)
The decorator for loading the user.

Parameters **callback** – The callback function that can load a user given a unicode `user_id`.

Returns The callback function

8.4 flask_blogging.processor module

class `flask_blogging.processor.`**PostProcessor**
Bases: `object`

classmethod all_extensions()

classmethod construct_url(*post*)

static create_slug(*title*)

classmethod custom_process(*post*)
Override this method to add additional processes. The result is that the `post` dict is modified or enhanced with newer key value pairs.

Parameters **post** (*dict*) – The post data with values for keys such as title, text, tags etc.

classmethod process(*post*, *render=True*)
This method takes the post data and renders it :param post: :param render: :return:

classmethod render_text(*post*)

classmethod set_custom_extensions(*extensions*)

8.5 flask_blogging.sqlastorage module

class `flask_blogging.sqlastorage.`**SQLAStorage**(*engine*, *table_prefix=''*)
Bases: *flask_blogging.storage.Storage*

The `SQLAStorage` implements the interface specified by the `Storage` class. This class uses SQLAlchemy to implement storage and retrieval of data from any of the databases supported by SQLAlchemy. This

__init__(*engine*, *table_prefix=''*)
The constructor for the `SQLAStorage` class.

Parameters **engine** – The `SQLAlchemy` engine instance created by calling

`create_engine`. One can also use Flask-SQLAlchemy, and pass the engine property. :type engine: object :param table_prefix: (Optional) Prefix to use for the tables created

(default `""`).

count_posts(*tag=None*, *user_id=None*, *include_draft=False*)
Returns the total number of posts for the give filter

Parameters

- **tag** (*str*) – Filter by a specific tag
- **user_id** (*str*) – Filter by a specific user
- **include_draft** (*bool*) – Whether to include posts marked as draft or not

Returns The number of posts for the given filter.

delete_post(*post_id*)
Delete the post defined by `post_id`

Parameters **post_id** (*int*) – The identifier corresponding to a post

Returns Returns True if the post was successfully deleted and False otherwise.

get_post_by_id(*post_id*)
Fetch the blog post given by `post_id`

Parameters **post_id** (*int*) – The post identifier for the blog post

Returns If the `post_id` is valid, the post data is retrieved, else returns `None`.

get_posts(*count=10*, *offset=0*, *recent=True*, *tag=None*, *user_id=None*, *include_draft=False*)
Get posts given by filter criteria

Parameters

- **count** (*int*) – The number of posts to retrieve (default 10)
- **offset** (*int*) – The number of posts to offset (default 0)
- **recent** (*bool*) – Order by recent posts or not
- **tag** (*str*) – Filter by a specific tag
- **user_id** (*str*) – Filter by a specific user
- **include_draft** (*bool*) – Whether to include posts marked as draft or not

Returns A list of posts, with each element a dict containing values for the following keys: (title, text, draft, post_date, last_modified_date). If count is `None`, then all the posts are returned.

save_post(*title*, *text*, *user_id*, *tags*, *draft=False*, *post_date=None*, *last_modified_date=None*, *meta_data=None*, *post_id=None*)
Persist the blog post data. If `post_id` is `None` or `post_id` is invalid, the post must be inserted into the storage. If `post_id` is a valid id, then the data must be updated.

Parameters

- **title** (*str*) – The title of the blog post
- **text** (*str*) – The text of the blog post
- **user_id** (*str*) – The user identifier
- **tags** (*list*) – A list of tags
- **draft** (*bool*) – (Optional) If the post is a draft of if needs to be published. (default `False`)
- **post_date** (*datetime.datetime*) – (Optional) The date the blog was posted (default datetime.datetime.utcnow())
- **last_modified_date** (*datetime.datetime*) – (Optional) The date when blog was last modified (default datetime.datetime.utcnow())
- **post_id** (*int*) – (Optional) The post identifier. This should be `None` for an insert call, and a valid value for update. (default `None`)

Returns The post_id value, in case of a successful insert or update. Return `None` if there were errors.

8.6 flask_blogging.storage module

class `flask_blogging.storage.`**Storage**
Bases: `object`

count_posts(*tag=None*, *user_id=None*, *include_draft=False*)
Returns the total number of posts for the give filter

Parameters

- **tag** (*str*) – Filter by a specific tag
- **user_id** (*str*) – Filter by a specific user
- **include_draft** (*bool*) – Whether to include posts marked as draft or not

Returns The number of posts for the given filter.

delete_post(*post_id*)
Delete the post defined by `post_id`

Parameters **post_id** (*int*) – The identifier corresponding to a post

Returns Returns True if the post was successfully deleted and False otherwise.

get_post_by_id(*post_id*)
Fetch the blog post given by `post_id`

Parameters **post_id** (*int*) – The post identifier for the blog post

Returns If the `post_id` is valid, the post data is retrieved,

else returns `None`.

get_posts(*count=10*, *offset=0*, *recent=True*, *tag=None*, *user_id=None*, *include_draft=False*)
Get posts given by filter criteria

Parameters

- **count** (*int*) – The number of posts to retrieve (default 10). If count is `None`, all posts are returned.
- **offset** (*int*) – The number of posts to offset (default 0)
- **recent** (*bool*) – Order by recent posts or not
- **tag** (*str*) – Filter by a specific tag
- **user_id** (*str*) – Filter by a specific user
- **include_draft** (*bool*) – Whether to include posts marked as draft or not

Returns A list of posts, with each element a dict containing values for the following keys: (title, text, draft, post_date, last_modified_date). If count is `None`, then all the posts are returned.

static normalize_tags(*tags*)

save_post(*title*, *text*, *user_id*, *tags*, *draft=False*, *post_date=None*, *last_modified_date=None*, *meta_data=None*, *post_id=None*)
Persist the blog post data. If `post_id` is `None` or `post_id` is invalid, the post must be inserted into the storage. If `post_id` is a valid id, then the data must be updated.

Parameters

- **title** (*str*) – The title of the blog post
- **text** (*str*) – The text of the blog post
- **user_id** (*str*) – The user identifier
- **tags** (*list*) – A list of tags
- **draft** (*bool*) – If the post is a draft of if needs to be published.
- **post_date** (*datetime.datetime*) – (Optional) The date the blog was posted (default datetime.datetime.utcnow())
- **last_modified_date** (*datetime.datetime*) – (Optional) The date when blog was last modified (default datetime.datetime.utcnow())
- **meta_data** (*dict*) – The meta data for the blog post
- **post_id** (*int*) – The post identifier. This should be `None` for an insert call, and a valid value for update.

Returns The post_id value, in case of a successful insert or update.

Return `None` if there were errors.

8.7 flask_blogging.views module

`flask_blogging.views.`**delete**(**args*, ***kwargs*)

`flask_blogging.views.`**editor**(**args*, ***kwargs*)

`flask_blogging.views.`**index**(*count*, *page*)

Serves the page with a list of blog posts

Parameters

- **count** –
- **offset** –

Returns

`flask_blogging.views.`**page_by_id**(*post_id*, *slug*)

`flask_blogging.views.`**posts_by_author**(*user_id*, *count*, *page*)

`flask_blogging.views.`**posts_by_tag**(*tag*, *count*, *page*)

`flask_blogging.views.`**recent_feed**()

`flask_blogging.views.`**sitemap**()

8.8 flask_blogging.forms module

class `flask_blogging.forms.`**BlogEditor**(*formdata=<class flask_wtf.form._Auto>*, *obj=None*, *prefix=''*, *csrf_context=None*, *secret_key=None*, *csrf_enabled=None*, **args*, ***kwargs*)

draft = <UnboundField(BooleanField, ('draft',), {'default': False})>

submit = <UnboundField(SubmitField, ('submit',), {})>

tags = <UnboundField(StringField, ('tags',), {'validators': [<wtforms.validators.DataRequired object at 0x7efd26e48490

text = <UnboundField(TextAreaField, ('text',), {'validators': [<wtforms.validators.DataRequired object at 0x7efd26e484

title = <UnboundField(StringField, ('title',), {'validators': [<wtforms.validators.DataRequired object at 0x7efd26e4839

CHAPTER 9

Contributors

- Gouthaman Balaraman
- adilosa

f

Symbols

A

B

C

D

E

F

G

I

N

P

R

S

T

U